ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Supporti Runtime ad Alte Prestazioni per WebAssembly: il Caso di WasmEdge

Relatore
prof. Paolo Bellavista

Laureando
Lorenzo Pellegrino

II sessione a.a. 2022/2023

Indice

Elenco delle figure

Introduzione

L'evoluzione delle tecnologie Web, spinta dalla crescente domanda di servizi online, ha portato alla proliferazione di dispositivi connessi. Affrontare la gestione senza precedenti dei dati richiede tecnologie efficienti e sicure. In questo contesto, il cloud computing emerge come risposta, rivoluzionando la distribuzione di servizi e garantendo scalabilità per soddisfare le esigenze degli utenti. La scalabilità e la disponibilità di un servizio diventano cruciali per la sua erogazione, qualunque esso sia. A tal scopo è fondamentale la scelta di soluzioni efficienti e rapide sia server-side che client-side, in quanto la rapidità e la velocità con cui il cloud computing eroga i servizi è data dalle tecnologie che lo compongono. Una tecnologia emergente che cerca di rispondere a queste richieste è WebAssembly, un formato binario ad alte prestazioni già abbastanza consolidato nel browser. Negli ultimi anni però la sua efficienza ha spinto gli sviluppatori a portare le sue prestazioni fuori dal browser tramite WASI e tramite la realizzazione di numerosi motori di esecuzione Wasm. In particolare, ci si concentrerà su un runtime emergente come WasmEdge.

Nel corso della tesi poi analizzeremo il motore da vicino, seguendone il flusso interno di esecuzione. Andremo anche a individuare i componenti nel codice sorgente, per capire come questi interagiscono tra di loro e come insieme permettano l'esecuzione di codice WebAssembly. Vedremo poi come il suo funzionamento si riflette sulla sua interfaccia di utilizzo. In seguito poi andremo a collocarlo nell'ecosistema della

distribuzione di micorservizi tramite Docker e analizzandone i principali casi d'uso.

Infine, WasmEdge verrà comparato con un runtime estremamente utilizzato al giorno d'oggi, come Node.js, e verrà anche comparato con altri runtime Wasm. Il confronto si fonderà su due prototipi, in JavaScript e Rust. Questi, con l'idea di simulare a grandi linee un'applicazione generica, svolgeranno le stesse operazioni di lettura/scrittura di file e di elaborazione, al fine di evidenziare le sole differenze di prestazioni. Per categorizzare i risultati, verranno usati diversi ordini di grandezza di carichi in input e verranno usati diverse operazioni computazionalmente onerose.

Capitolo 1

WasmEdge: Obiettivi e Motivazioni della Tesi

In questa tesi, esploreremo WasmEdge, una tecnologia emergente destinata a guadagnare sempre più rilevanza nell'ambito delle applicazioni basate su WebAssembly (Wasm). Tuttavia, per acquisire una comprensione completa dei suoi vantaggi e del suo funzionamento, inizieremo fornendo una panoramica delle tecnologie correlate. Questa introduzione ci permetterà di contestualizzare WasmEdge e di comprendere il suo stretto rapporto con le fondamenta su cui si basa.

1.1 Contesto

Nel corso degli ultimi decenni, l'evoluzione delle tecnologie Web ha segnato un notevole progresso, spinta dalla crescente domanda di applicazioni e servizi online e dalla costante innovazione tecnologica. Questa crescita ha portato a una proliferazione di dispositivi connessi a Internet, aprendo le porte a un accesso sempre più

diffuso ai servizi Web. Di conseguenza, è diventato imperativo implementare tecnologie più efficienti, sicure e veloci per soddisfare le esigenze degli utenti e garantire la disponibilità continua dei servizi.
La quantità di dati gestita è senza precedenti, con migliaia di informazioni generate quotidianamente attraverso una vasta gamma di servizi utilizzati nella vita di tutti i giorni. In questo contesto, l'evoluzione tecnologica si è rivelata essenziale per far fronte a questa crescente domanda di risorse e garantire un accesso affidabile e immediato ai servizi online.
Gli utenti cercano costantemente nuove funzionalità, maggiore potenza di calcolo e prestazioni ottimali. Con un aumento costante nella quantità di dati personali in circolazione, diventa essenziale che gli sviluppatori e i fornitori di servizi implementino misure di sicurezza rigorose. Il continuo aumento di funzionalità e complessità delle applicazioni comporta anche notevoli disagi in caso di interruzioni della connessione o di latenza eccessiva.
In risposta a queste sfide, ha preso vita il cloud computing, una rivoluzione nell'ambito delle tecnologie Web che ha trasformato la distribuzione di servizi e applicazioni. Il concetto alla base del cloud computing è quello di fornire risorse informatiche, come server, storage, database e reti, attraverso Internet. Il cloud computing deve garantire un'erogazione flessibile e accessibile, che possa fornire la massima scalabilità possibile.
La scalabilità è un concetto cruciale nel contesto del cloud computing, rappresentando la capacità di un sistema di adattarsi alle esigenze in costante evoluzione, sia per crescere che per ridursi in dimensioni in risposta al carico di lavoro. La scalabilità è fondamentale per garantire che le applicazioni e i servizi possano operare senza soluzione di continuità, mantenendo alte prestazioni e soddisfacendo le aspettative degli utenti. Questo è comunemente misurato dalla percentuale di disponibilità, che calcola il tempo durante il quale i servizi sono erogati correttamente (uptime)

rispetto al tempo atteso (tempo di osservazione).

$$Disponibilità = \frac{Uptime}{Osservazione} \tag{1.1}$$

disponibilità %	downtime per anno	downtime per mese	downtime per settimana
98%	7,3 giorni	14,4 ore	3,36 ore
99%	3,65 giorni	7,20 ore	1,68 ore
99,5%	1,83 giorni	3,60 ore	50,4 minuti
99,9%	8,76 ore	43,2 minuti	10,1 minuti
99,99%	52,6 minuti	4,32 minuti	1,01 minuti
99,999%	5,26 minuti	25,9 secondi	6,05 secondi
99,9999%	31,5 secondi	2,59 secondi	0,605 secondi

Figura 1.1: L'ordine di grandezza delle disponibilità [15]

1.2 Evoluzione tecnologica

Parallelamente all'evoluzione descritta in precedenza, le architetture dei sistemi informatici hanno anch'esse attraversato quattro principali fasi di sviluppo.

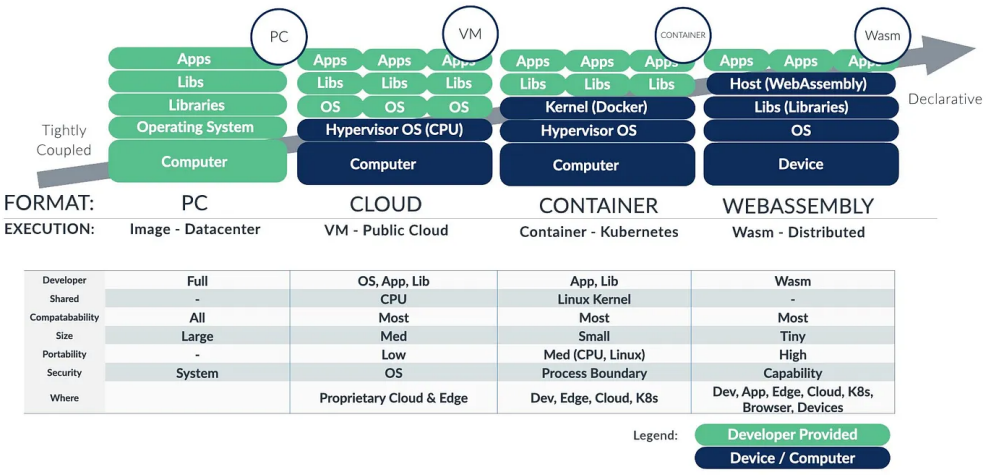

Figura 1.2: L'evoluzione delle infrastrutture informatiche [18]

1.2.1 PC (Computer Fisici)

Nei computer fisici, le risorse sono dedicate a singole applicazioni. Questo significa che ciascun'applicazione ha accesso a una quantità specifica di memoria fisica e funziona su un sistema operativo con le relative librerie. Tuttavia, questa configurazione limita la flessibilità delle applicazioni poiché le risorse fisiche sono statiche. In altre parole, se un'applicazione richiede risorse aggiuntive a causa di un aumento del carico di lavoro, la struttura fisica potrebbe non essere pronta ad affrontare questa richiesta. La conseguenza è che potrebbe essere necessario investire tempo e denaro nell'acquisto e nella configurazione di nuovi computer per far fronte alla crescente domanda.
Questa mancanza di adattabilità diventa particolarmente problematica quando l'aumento del carico di lavoro non è costante e prolungato nel tempo, poiché gli investimenti iniziali non vengono completamente ammortizzati. In altre parole, la scalabilità è molto limitata in un ambiente basato su dispositivi fisici. Inoltre, i dispositivi fisici sono soggetti a guasti hardware occasionali, che possono causare interruzioni del servizio. Per minimizzare tali interruzioni, è necessaria manutenzione e gestione, attività che sono distinte dallo sviluppo delle applicazioni.
Quando si eseguono applicazioni su dispositivi fisici, queste condividono le risorse con altre applicazioni. Questo può portare a problemi di isolamento, dove le applicazioni potrebbero interferire l'una con l'altra, causando conflitti e problemi nell'erogazione dei servizi.

1.2.2 VM (Macchine Virtuali)

Le macchine virtuali (VM) sfruttano un componente software chiamato **Hypervisor**, il quale, una volta installato su una macchina fisica (host), crea un ambiente in cui possono essere eseguiti più sistemi operativi in modo isolato. L'Hypervisor gestisce le risorse del sistema in modo da suddividerle ed ottimizzarle, garantendo

che ciascuna VM non sia consapevole delle altre VM sullo stesso host, eliminando così i conflitti di cui si è parlato in precedenza.
Oltre a permettere l'esecuzione di più sistemi operativi in modo isolato, l'Hypervisor utilizza la virtualizzazione dell'hardware per emulare l'hardware del computer. In questo modo, ogni VM ha un ambiente virtuale che assomiglia a una macchina fisica indipendente. Ciò consente di eseguire diverse applicazioni sulla stessa macchina fisica, agevolando lo sviluppo delle applicazioni su diversi sistemi operativi.
Un aspetto importante è la capacità di esportare e importare le VM tra host, il che fornisce protezione dalle guasti hardware. Questo processo avviene attraverso l'utilizzo di immagini che rappresentano uno snapshot completo della VM, inclusi il sistema operativo, le applicazioni e i dati.
Tuttavia, è cruciale notare che le VM richiedono una quantità significativa di risorse, che può variare da diverse centinaia di MB a diversi GB, a causa della presenza dell'intero sistema operativo nell'immagine. Inoltre, l'avvio delle VM è generalmente più lento rispetto all'avvio su hardware nativo a causa della virtualizzazione. Quindi, sebbene la scalabilità delle VM sia superiore rispetto ai PC tradizionali, l'overhead introdotto dai sistemi operativi e la relativa lentezza all'avvio ne limitano l'utilizzo in scenari ad alta mole di traffico.

1.2.3 Container

I container sono unità di esecuzione leggere, portatili e autosufficienti che racchiudono un'applicazione insieme a tutte le sue dipendenze, compresi file di sistema, librerie e variabili d'ambiente, all'interno di un'immagine dedicata. Questo è possibile grazie a un **Container Engine** che interagisce con il kernel dell'host e si occupa della creazione, condivisione e esecuzione delle immagini.
La comunicazione tra i container è consentita, ma avviene solo attraverso una rete virtuale definita dal Container Engine, garantendo così l'isolamento dei container a

meno che non siano esplicitate richieste di connessione a un livello più basso. Questo isolamento protegge gli altri container e l'host da potenziali conflitti. Inoltre, il Container Engine fornisce strumenti per monitorare le prestazioni dei container.
È importante notare che ogni container non include un sistema operativo completo, grazie alla collaborazione tra il Container Engine e il kernel dell'host. Questo si traduce in un consumo di risorse significativamente inferiore rispetto alle macchine virtuali (VM), che può variare da decine a centinaia di megabyte.
Questa caratteristica facilita e velocizza lo sviluppo delle applicazioni, aumentando l'efficienza e la flessibilità per i clienti. Tuttavia, a causa della condivisione del kernel dell'host, i container potrebbero comportare problemi di sicurezza e funzionamento generale se il kernel dell'host venisse compromesso. Nonostante ciò, i container offrono una maggiore scalabilità ed efficienza rispetto alle VM.

1.2.4 WASM (+ WASI)

Pur consentendo alte prestazioni, i container ancora richiedono che le applicazioni siano con tutte le loro dipendenze e spesso compilate in linguaggi diversi. Tuttavia, negli ultimi anni è emersa una tecnologia innovativa chiamata WebAssembly (WASM) che sta guadagnando sempre più popolarità.
WASM è stata sviluppata con l'obiettivo di migliorare le prestazioni lato browser, superando le limitazioni associate a JavaScript (JS). Utilizza un formato binario precompilato che è più leggero e più facilmente trasportabile. Inoltre, è notevolmente più veloce poiché non richiede il processo di compilazione e non è afflitto dai problemi di compatibilità hardware o software.
WASM implementa anche un meccanismo di sandboxing che riduce la superficie di attacco della memoria, contribuendo a rendere l'esecuzione delle applicazioni più sicura. Più recentemente, è stata introdotta la WebAssembly System Interface (WASI), un'interfaccia per WASM con lo scopo di supportare lo sviluppo di applicazioni fuori dal browser.

WASI eredita le caratteristiche di WASM e le sfrutta per supportare lo sviluppo di applicazioni server-side. Risolve il problema della sicurezza mediante l'uso di sandboxing, mantenendo al contempo elevate prestazioni.
Questi concetti saranno rilevanti mentre esploreremo in dettaglio WASM e il suo ecosistema in seguito.

1.3 WASM più nel dettaglio

Nella pratica, WebAssembly[20] è un formato binario con estensione .wasm, progettato per essere eseguito da una macchina virtuale incorporata all'interno dei browser. È stato appositamente creato per essere più efficiente, veloce e sicuro rispetto a JavaScript, che è un linguaggio interpretato e a tipizzazione dinamica. Questa caratteristica lo rende particolarmente adatto per operazioni altamente complesse, come il rendering video e le grafiche in 3D, dove JavaScript potrebbe causare notevoli rallentamenti.
L'essere un formato binario a basso livello (simile all'Assembly) consente al processore dell'host di eseguirlo direttamente, senza ulteriori elaborazioni o conversioni. Questo è il motivo principale per cui WebAssembly è noto per le sue prestazioni notevolmente veloci.

1.3.1 Produzione di un file .wasm

L'esecuzione di un file binario WebAssembly rappresenta l'ultimo passo di un processo di produzione composto da diverse fasi[3]:

1. **Scrittura**

   La scrittura del codice avviene in uno dei linguaggi che supportano WebAssembly. In figura 1.3 si riportano alcuni tra i linguaggi più famosi[1] e il loro

[1] https://www.fermyon.com/wasm-languages/webassembly-language-support

rapporto con Wasm.

2. **Compilazione**

   WebAssembly non è un linguaggio di programmazione autonomo, ma il risultato di un processo di compilazione. Questo processo comporta la conversione del codice sorgente in un file binario eseguibile. Il file binario contiene istruzioni a basso livello comprensibili dalla CPU del computer.

3. **Import**

   Tuttavia, il file binario non può essere eseguito fino a quando non viene importato in un browser Web. A questo punto, il motore del browser può decodificare, compilare e tradurre il file in codice eseguibile per il computer del cliente.

4. **Istanziazione**

   Il motore JavaScript crea un'istanza virtuale isolata, proprio come fa con i file JavaScript. All'interno di questa istanza, il file binario WebAssembly importato può essere istanziato ed eseguito.

Come è evidente da questo processo, WebAssembly supera le limitazioni di JavaScript senza però avere l'obiettivo di sostituirlo. Piuttosto, WebAssembly sfrutta JavaScript per la propria esecuzione. JavaScript rimarrà la scelta preferita per l'esecuzione di codice non ad alte prestazioni.

Linguaggio	Browser	WASI
JavaScript	Si	Work in progress
Python	Work in progress	Si
Java	Si	Si
PHP	Si	Si
C / .NET	Si	Si
C++	Si	Si
C	Si	Si
Typescript	Work in progress	No
Ruby	Si	Si
Go	Si	Si
Rust	Si	Si

Figura 1.3: Alcuni dei linguaggi che supportano Wasm

1.3.2 Funzionamento di un file .wasm

Ogni file con estensione .wasm contiene un **modulo** WebAssembly autonomo o fa parte di un modulo più ampio. Ogni modulo è suddiviso in due sezioni: la sezione di importazione e la sezione di definizione.
Nella sezione di importazione, vengono elencate le funzioni, le variabili e altri elementi che vengono importati da altri moduli. La sezione di definizione, invece, elenca le funzioni e le variabili definite all'interno del modulo stesso. Le funzioni vengono poi riferite in apposite strutture dati chiamate **tabelle**.
Le funzioni sono chiamate tramite istruzioni, che sono serie di comandi simili a quelli dei linguaggi assembly. Queste istruzioni vengono eseguite da un motore WebAssembly. Oltre a chiamare funzioni, possono anche eseguire operazioni aritmetiche, logiche e di controllo del flusso.
Tutte queste operazioni vengono eseguite in modalità **sandboxed**, il che significa che loro viene riservata un'area di memoria circoscritta. Questo previene danni a parti di memoria destinate ad altri scopi, garantendo l'isolamento e la sicurezza

dell'esecuzione.

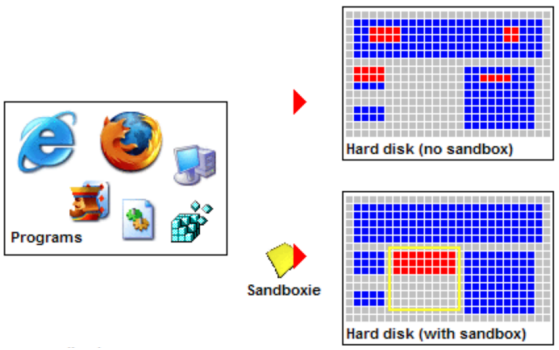

Figura 1.4: Scrittura su una memoria senza sandbox e con sandbox. [16]

La **memoria** in WebAssembly è rappresentata da un array tipato in JavaScript. In questi casi, si fa spesso riferimento al 'codice collante' o 'glue-code', che agisce come un ponte tra WebAssembly, che utilizza dati fortemente tipizzati, e JavaScript, noto per la sua flessibilità e dinamicità.
L'allocazione della memoria in WebAssembly avviene in modo più preciso rispetto a JavaScript. Ogni dato in WebAssembly deve avere un tipo ben specifico, il che garantisce che venga allocata solo la quantità di memoria strettamente necessaria e che non ci siano sprechi di risorse.

1.4 WASI più nel dettaglio

All'interno del browser, WebAssembly funziona come intermediario con il sistema operativo, e questo processo generalmente non presenta problemi significativi. Tuttavia, l'efficienza di WebAssembly è così importante che oggi si sta cercando di ottenere le stesse prestazioni al di fuori del browser.
È proprio per questo scopo che è stata appositamente progettata WASI.

1.4.1 Interfaccia di sistema

Un'interfaccia di sistema definisce uno standard a cui gli sviluppatori possono fare riferimento senza preoccuparsi del sistema operativo sottostante. Sarà il compilatore a scegliere l'implementazione specifica dell'interfaccia.
Grazie a un'interfaccia standardizzata come WASI (WebAssembly System Interface), le applicazioni WebAssembly possono accedere alle risorse di sistema, come file, dispositivi di rete, input/output e altro. Questo significa che un'applicazione WebAssembly che utilizza le chiamate di sistema WASI può essere compilata una volta e poi eseguita su diversi ambienti che supportano WASI, senza la necessità di adattarla specificamente per ciascun ambiente.

1.4.2 Accenno sul funzionamento

Per consentire l'accesso al sistema operativo sottostante, WASI implementa un insieme di operazioni dello standard POSIX. Questo significa che le chiamate di sistema vengono intercettate e viene eseguita la versione implementata su WASI. Inoltre, queste implementazioni considerano le regole sugli accessi ai file definite in precedenza, migliorando notevolmente la sicurezza, come vedremo in dettaglio successivamente.

1.4.3 Sicurezza

Nel contesto di WASI, la sicurezza si basa sul concetto di 'Capability-Based Security'. Questo approccio si fonda sul concetto di 'capability', che rappresenta un token di autorità che è comunicabile e non falsificabile. Ogni processo riceve una serie di capability.
Il concetto può essere implementato in diverse modalità, adattandosi alle esigenze specifiche di ciascun caso d'uso. In generale, si tratta di un'associazione tra il nome della risorsa e una lista di permessi relativi a quella risorsa.

In pratica, ogni volta che un processo tenta di accedere a una risorsa, il sistema controlla le capability a sua disposizione per quella risorsa specifica per verificare se dispone dei permessi necessari. In tal modo, il sistema operativo non deve verificare gli accessi a ciascuna risorsa richiesta dal processo, ma piuttosto verifica le capability che il processo stesso porta con sé, migliorando notevolmente l'efficienza e la sicurezza del sistema.

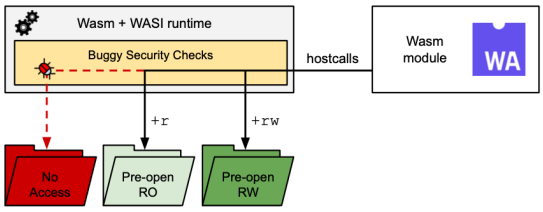

Figura 1.5: Esempio di come funziona l'accesso ai file con WASI [1]

1.5 Concetto di runtime

Una volta ottenuto il file eseguibile, è fondamentale introdurre il concetto di 'runtime'. Questo rappresenta un ambiente di esecuzione essenziale per un programma o un'applicazione. Esso fornisce le risorse e le funzionalità necessarie affinché il software possa funzionare correttamente. In generale, il runtime si occupa di gestire la memoria, le chiamate alle funzioni, la gestione delle risorse hardware e altre attività necessarie per l'esecuzione del programma.
Ogni linguaggio di programmazione dispone del proprio runtime. Ad esempio, Java ha la JVM (Java Virtual Machine), JavaScript ha Node.js, Python ha il Python Runtime, e così via.

1.6 Introduzione a WasmEdge

WebAssembly dispone di diversi runtime[2], tra cui Wasmer, Wasmtime e WasmEdge, oltre ad altri. Quest'ultimo è quello su cui ci focalizzeremo, osservandone nel dettagliola struttura e il funzionamento, contestualizzandolo nei suoi principali casi d'uso. WasmEdge è stato progettato per l'esecuzione di codice WebAssembly su una varietà di piattaforme, offrendo un ambiente di runtime ottimizzato per applicazioni a bassa latenza e altamente portabili. Le sue caratteristiche di sicurezza e l'ampia compatibilità lo rendono una scelta popolare per le applicazioni che richiedono l'esecuzione veloce e sicura di codice WebAssembly.

1.7 Introduzione al prototipo

Più avanti, presenteremo un prototipo progettato per evidenziare le differenze di prestazioni tra WebAssembly e JavaScript, focalizzandoci su aspetti l'I/O di file e l'utilizzo della CPU.
Il prototipo eseguirà operazioni specifiche e sarà sottoposto a stress a diversi livelli, consentendo di raccogliere dati significativi per poi visualizzare le misurazioni di interesse. Sarà implementato sia in JavaScript, eseguito tramite Node.js, sia in Rust. Con quest'ultimo verrà creato il binario WebAssembly, che sarà eseguito utilizzando runtime come WasEdge, Wasmtime e Wasmer.
In questo modo, forniremo una panoramica dettagliata non solo delle differenze tra WebAssembly e JavaScript, ma anche delle disparità tra i principali runtime WebAssembly.

[2] https://github.com/appcypher/awesome-wasm-runtimes

Capitolo 2

WasmEdge

Dopo aver introdotto in generale l'ambiente in cui nasce WasmEdge, andiamo ad approfondire le caratteristiche specifiche dello strumento. Ci concentreremo sull'analizzare la struttura e il funzionamento di quest'ultimo, per poi collocarlo all'interno dell'erogazione di servizi software, al fine di esplorare i suoi casi d'uso.

2.1 Compilatore

Per ottenere una visione più chiara del funzionamento di WasmEdge, è fondamentale esaminare in dettaglio i compilatori per capire quale sia il loro ruolo e come funzionano. Un compilatore è un programma informatico che traduce una serie di istruzioni scritte in un determinato linguaggio di programmazione (codice sorgente) in istruzioni di un altro linguaggio (codice oggetto) e questo processo di traduzione si chiama **compilazione**.

2.1.1 Accenno ai compilatori

Durante il processo di compilazione, è possibile distinguere tre fasi chiave:

1. **Analisi** o front-end

   (a) **Analisi lessicale**

   Il codice sorgente viene letto carattere per carattere, eliminando spazi e commenti per ottenere una forma più strutturata.

   (b) **Analisi sintattica**

   Il codice sorgente viene analizzato per identificare la struttura grammaticale del linguaggio di programmazione. Viene quindi creato l'AST (Abstract Syntax Tree), ovvero una rappresentazione gerarchica del codice che mette in evidenza la relazione tra le espressioni, le dichiarazioni, i blocchi di codice, le funzioni e altre costruzioni linguistiche del linguaggio.

   (c) **Analisi semantica**

   Questa fase verifica la coerenza semantica del codice, assicurando che le operazioni siano valide (variabili dichiarate, tipi coerenti, ecc.).

2. **Generazione di IR** (Intermediate Representation)

   SI tratta di un'astrazione del codice sorgente ed è spesso indipendente dall'architettura del computer di destinazione. È una rappresentazione strutturata del codice sorgente, quindi mantiene tutte le relazioni ottenute dall'AST. È progettato in modo che le ottimizzazioni del codice possano essere applicate in modo più agevole rispetto al codice sorgente. Questo consente al compilatore di eseguire una serie di trasformazioni che migliorano le prestazioni e la leggibilità del codice prima di generare il codice target.

3. **Sintesi** o back-end

   (a) **Ottimizzazione**

   Questa fase sfrutta l'astrazione dell'IR per individuare più facilmente come migliorare le prestazioni, ridurre la dimensione del codice e semplificare la sua struttura.

(b) **Generazione di MC** (Machine Code)

In questa fase abbiamo la generazione del codice macchina specifico per l'architettura di destinazione e sarà ciò che verrà eseguito direttamente sulla macchina target.

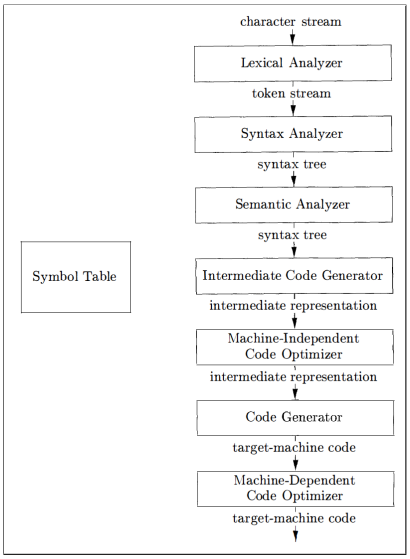

Figura 2.1: Processo di compilazione

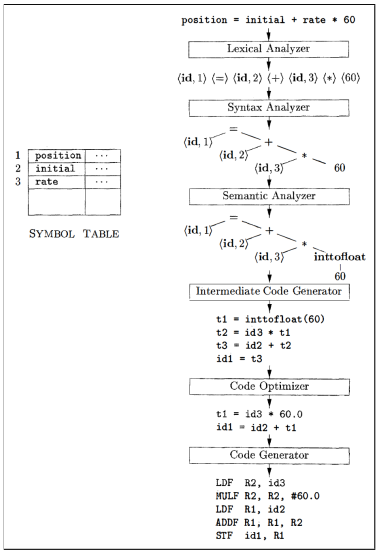

Figura 2.2: Esempio di compilazione [2]

2.2 LLVM (Low Level Virtual Machine)

LLVM[19] rappresenta un insieme di compilatori e strumenti correlati con un focus specifico su aspetti come l'analisi, la trasformazione e l'ottimizzazione del codice. Questo framework è noto per la sua elevata flessibilità e modularità e fornisce implementazioni per le diverse fasi che abbiamo appena esaminato.

L'LLVM MC (Machine Code) è supportato da numerose architetture, tra cui IA-32, x86-64, ARM, ARM64 e molte altre[1], il che lo rende adatto per compilare una vasta gamma di codice sorgente. Inoltre, LLVM consente di selezionare il livello

[1] https://llvm.org/Features.html

di ottimizzazione desiderato durante il processo di compilazione. Un livello di ottimizzazione più elevato può migliorare le prestazioni a discapito di una maggiore complessità del codice.

2.3 JIT vs AOT

Sono due strategie di compilazione:

1. **JIT** (Just In Time): In questa strategia, il compilatore traduce il codice durante l'esecuzione del programma. La traduzione in codice macchina avviene appena prima dell'esecuzione delle istruzioni.

2. **AOT** (Ahead Of Time): In questa strategia, il compilatore traduce il codice in codice macchina prima dell'esecuzione del programma.

	JIT	AOT
Adattabilità	Il compilatore può adattarsi alle condizioni effettive di esecuzione del programma, ottenendo migliori ottimizzazioni.	Il codice è compilato in anticipo, abbiamo quindi una scarsa adattabilità che può portare a prestazioni subottimali in alcune situazioni.
Overhead	Comporta un overhead iniziale, poiché è necessario avviare il processo di compilazione prima dell'esecuzione.	Poiché il codice è già stato compilato, non c'è alcun overhead iniziale al momento dell'avvio.
Portabilità	È spesso più portabile, in quanto il compilatore può generare codice specifico per l'architettura della macchina in cui viene eseguito.	È poco portabile perché, essendo ottimizzato, richiede un'attenta pianificazione e considerazione delle esigenze specifiche dell'applicazione e delle piattaforme di destinazione.
Memoria	Il codice JIT generato deve essere memorizzato in memoria, il che può aumentare il consumo di memoria del processo.	Il codice AOT è più compatto e quindi più leggero, quindi abbiamo meno codice da caricare in memoria.
Sicurezza	Poiché il codice viene compilato durante l'esecuzione, potrebbe esistere il rischio di vulnerabilità di sicurezza legate ad attacchi come l'iniezione di codice malevolo.	Il codice viene compilato fuori dall'esecuzione, e quindi evidenzierebbe le iniezioni di codice senza che queste possano agire.

Figura 2.3: Confronto tra JIT e AOT

Quello che possiamo dedurre da ciò è che l'approccio AOT permette un'esecuzione molto più veloce e un avvio rapido delle applicazioni. Questo è particolarmente utile nel contesto descritto nell'introduzione, in cui si richiedono prestazioni elevate e un accesso immediato ai servizi.

2.4 Come funziona internamente WasmEdge

Di seguito andremo nel dettaglio del funzionamento interno di WasmEdge, analizzandone il flusso di esecuzione.[4].

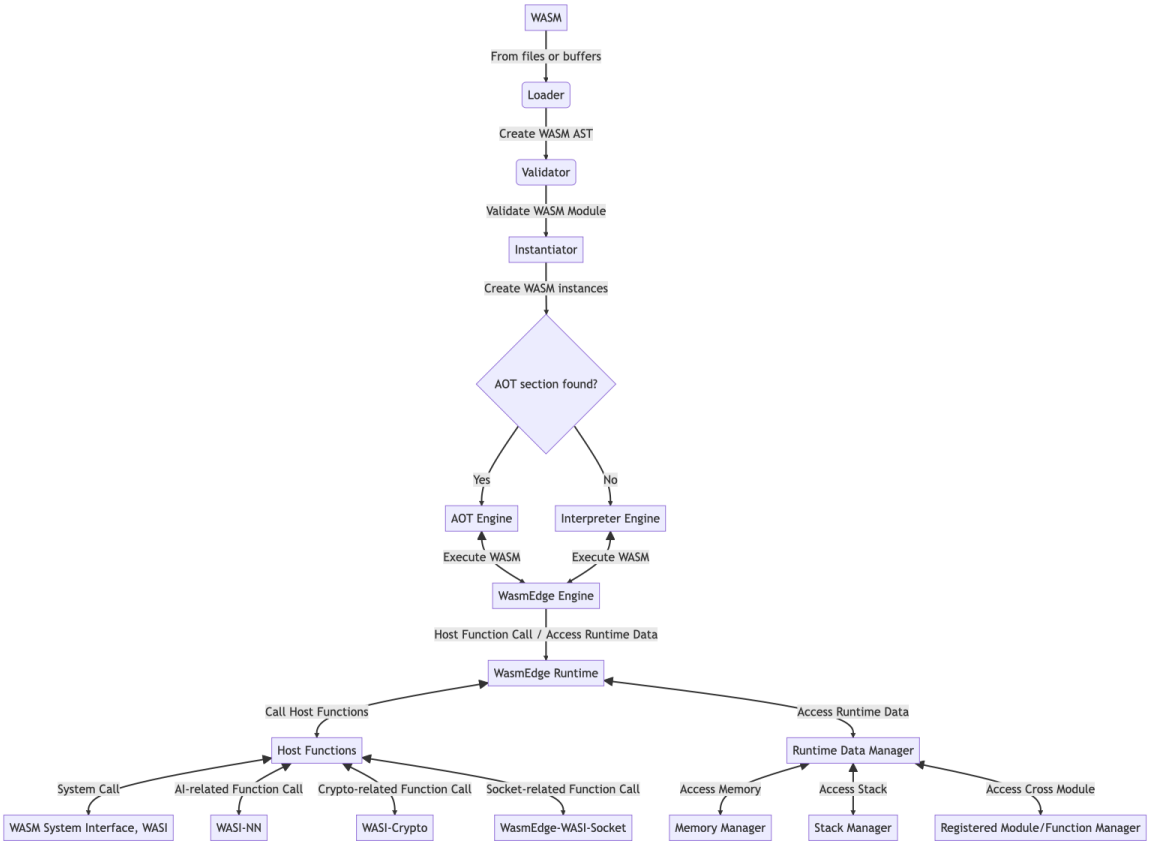

Figura 2.4: Il flusso che segue ogni esecuzione di WasmEdge

[21]

2.4.1 Analisi dei componenti

Di seguito faremo riferimento al codice sorgente, scaricabile anche da GitHub tramite il terminale con `git clone https://github.com/WasmEdge/docs` . Nel corso della lettura del codice sorgente, diventeranno evidenti delle differenze tra il modello (lo schema precedentemente illustrato) e la sua implementazione. Esaminiamo più da vicino i componenti:

1. **Loader** (`WasmEdge/include/loader/loader.h`)

   - `loadFile()`: consente di caricare il file .wasm in memoria, recuperandolo da una sorgente esterna. Il risultato di questa operazione è un array di byte con il contenuto del file.
   - `parseModule()`: permette di analizzare il modulo, prendendo in esame il vettore di byte creato in precedenza e procedendo con l'interpretazione. Durante questa fase, vengono estratti i tipi, le funzioni, le tabelle e altri elementi al fine di creare l'AST.
   - `reset()`: permette di pulire il buffer di memoria una volta finita l'operazione.

   Inoltre, è interessante notare che, già in questa fase, durante l'implementazione del modulo (nello specifico, in `WasmEdge/lib/loader/ast/module.cpp`), è presente una sezione che verifica e notifica se è presente o meno la sezione AOT. Questo controllo viene eseguito sullo stesso modulo in entrambi i casi. Inoltre, come vedremo in seguito, sarà possibile forzare l'interpretazione del file anche se la sezione AOT è presente.

2. **Validator** (`WasmEdge/include/validator/validator.h`)
   Ha il compito di controllare la correttezza semantica di un modulo durante il processo di caricamento. L'unico metodo pubblico è:

- `validate()`: richiama una serie di metodi privati che si occupano di validare le parti specifiche del modulo come i tipi, i segmenti, le descrizioni, le sezioni e le espressioni costanti. Da notare che questo metodo solleva eccezioni se fallisce.

3. **AOT Engine** (`WasmEdge/include/aot/*`)
   All'interno di questa cartella sono presenti quattro librerie:

   (a) `blake3.h`: si occupa dell'hashing dei dati (con algoritmo Blake3[2]). Ciò è utile per la generazione rapida di chiavi hash per garantire la sicurezza delle informazioni tramite un'identificazione dei moduli, delle cache e quindi per garantire anche l'integrità dei dati.

   (b) `cache.h`: si occupa della cache, in cui vengono inseriti i moduli compilati.

   (c) `compiler.h`: definisce la classe che si occupa di compilare il modulo wasm e di restituire un eseguibile binario. La sua implementazione fa uso di LLVM.

   (d) `version.h`: tiene traccia della versione binaria del progetto.

4. **Interpreter Engine**
   Come è evidenziato nel file `WasmEdge/Changelog.md`, la classe Interpreter preesistente è stata integrata nell'Executor, insieme a tutti i suoi metodi. In effetti, l'Executor opererà in modalità interpretativa (JIT) per impostazione predefinita, a meno che il Loader non rilevi la presenza della sezione AOT e notifichi tale situazione.

5. **Executor** (`WasmEdge/include/executor/executor.h`)
   Realizza Instantiator e WasmEdge Runtime. Essendo un file molto fornito, si fornisce solo una panoramica dei metodi pubblici:

[2] https://rustrepo.com/repo/BLAKE3-team-BLAKE3

- metodi di **istanziazione**: istanziano e registrano moduli, tabelle, dati ed altri elementi logici.
- metodi di **registrazione**: prendono le istanze e le registrano per tenerne traccia durante l'esecuzione.
- metodi di **call**: chiamano direttamente o indirettamente una funzione all'interno del modulo wasm.
- metodi di **memoria**: gestiscono la memoria wasm. Possono aumentarla, copiarla, riempirla o inizializzarla.
- metodi di **tabella**: gestiscono la tabella dei riferimenti. Possono ottenere o impostare un riferimento, inoltre possono aumentarla, copiarla, riempirla o inizializzarla.
- metodi di **thread**: gestiscono i thread lanciati durante l'esecuzione. Possono svegliare i thread o metterli in attesa di un evento.
- `trap()`: gestisce le eccezioni o le interruzioni durante l'esecuzione del modulo wasm.
- `dataDrop()` : elimina una specifica istanza di dati.
- `preprare()` : è un metodo privato che prepara il contesto di esecuzione, ovvero i riferimenti alle memorie e alle variabili globali.

Come detto in precedenza, qui viene scelto se usare la modalità AOT o Interpreter.

6. **Host Functions** (`WasmEdge/include/runtime/hostfunc.h`)
   Serve a definire e implementare funzioni personalizzate che possono essere richiamate all'interno di un modulo. Per raggiungere questo obiettivo, nel file si stabilisce un tipo generico di funzione denominato 'FuncType' e un metodo generico 'run()'. Questa approccio ci consente di utilizzare funzioni personalizzate con firme diverse senza la necessità di scrivere codice ripetitivo. Alcune di queste sono:

- WASI (`WasmEdge/include/wasi/*`): Forniscono l'interfaccia tra i programmi WebAssembly e il sistema operativo sottostante, consentendo loro di accedere a funzionalità come l'orologio di sistema, il file system e l'ambiente del programma.
- WASI-NN: fornisce un'interfaccia per l'esecuzione di operazioni legate a reti neurali da parte di programmi WebAssembly.
- WASI-Crypto: fornisce un'interfaccia per la crittografia e la gestione delle chiavi.
- WASI-Socket: fornisce un'interfaccia per la creazione e l'uso delle socket

7. **Runtime Data Manager** (`WasmEdge/include/runtime/*`)

- **Memory Manager** (`WasmEdge/include/executor/engine/memory.ipp`)
  Non esiste un file denominato 'memorymgr.h' in quanto la gestione dell'accesso alla memoria viene effettuata direttamente all'interno dell'Executor. In questo contesto, le operazioni di 'load' e 'store' consentono la lettura e la scrittura nella memoria. Per verificare se un'operazione sulla memoria può essere effettuata con successo, viene calcolato l'Effective Address (EA) e confrontato con la memoria stessa.

$$EA = offsetDato + indirizzoBaseMemoria \tag{2.1}$$

  In questo modo si prevengono operazioni di scrittura/lettura al di fuori della 'memoria di riferimento' o sandbox, evitando potenziali problemi. È fondamentale sottolineare che effettuando operazioni di scrittura o lettura al di fuori della sandbox, potremmo accidentalmente manipolare dati utilizzati per altri scopi, causando potenzialmente danni di vasta portata.
- **Stack Manager** (`stackmgr.h`)
  Esso è principalmente composto da due vettori: il ValueStack e il FrameStack. Il primo funge da stack per i valori generati durante l'esecuzione,

memorizzando i valori intermedi e quelli prodotti dalle istruzioni. Il secondo è un array di frame, ciascuno dei quali contiene il contesto di esecuzione del modulo e viene utilizzato per gestire l'esecuzione delle funzioni. Questo componente fornisce metodi che garantiscono un accesso sicuro allo stack e prevengono l'insorgere di errori inaspettati.

- **Registered Module/Function Manager** (`storemgr.h`)
  Questo componente si occupa della gestione dei moduli e delle funzioni durante l'esecuzione. Fornisce metodi per registrare, ottenere e rimuovere i moduli. Utilizza una tabella interna per mantenere riferimenti univoci ai moduli e su questa tabella vengono eseguite le operazioni sopra citate. Inoltre, dispone di un mutex che consente l'accesso mutuamente esclusivo in scrittura e l'accesso concorrente in lettura ai moduli, garantendo un ambiente multithreading sicuro.

2.4.2 WasmEdge VM

Tutto ciò che abbiamo esaminato finora avviene all'interno di una macchina virtuale che viene generata appositamente per l'esecuzione del codice. Viene definita in `WasmEdge/include/vm/vm.h` e implementata in `WasmEdge/lib/vm/vm.cpp` . Questa classe gestisce l'intero flusso di esecuzione dei moduli Wasm, dalla registrazione dei moduli all'esecuzione. Per avere un quadro più chiaro del contesto, forniamo un elenco dei componenti della classe VM.
All'interno di questa classe, vengono mantenute le seguenti informazioni: configurazioni del runtime, statistiche del runtime, un mutex per il supporto del multithreading, un Loader, un Validator, un Executor, un puntatore al modulo Wasm, il modulo attualmente istanziato, un vettore di altri moduli, moduli di plug-in e uno StoreManager.
La VM è progettata con un modello a fasi che rappresentano le diverse tappe nel ciclo di vita di una macchina virtuale per l'esecuzione di moduli WebAssembly.

Infatti è presente un enumeratore `VMStage` che tiene conto della fase corrente e funge da controllo affinché si possano effettuare solo le operazione permesse in una determinata fase. Le fasi sono:

1. `VMStage::Inited`, in cui la macchina virtuale è appena stata creata.

2. `VMStage::Loaded`, qui la macchina virtuale carica il modulo Wasm da una fonte esterna, che può essere un file sul disco, un array di byte o un oggetto `AST:Module`.

3. `VMStage::Validated`, dopo il caricamento del modulo, la macchina virtuale deve validare il modulo Wasm per assicurarsi che rispetti le specifiche del linguaggio WebAssembly.

4. `VMStage::Instantiated`, una volta validato, il modulo può essere istanziato, creando un'istanza eseguibile del modulo Wasm. L'istanziamento può coinvolgere la risoluzione di simboli, l'allocazione di memoria e l'inizializzazione delle variabili globali.

Dopo queste fasi avviene l'esecuzione tramite l'Executor che riceve i parametri e il codice, esegue quest'ultimo e restituisce il risultato tramite `Executor::invoke()`. Infine avviene la pulizia o 'cleanup', in cui VMStage viene riportato ad Inited e vengono resettate tutte le strutture dati della VM.
I metodi forniti dalla classe, sincronizzati da un funzionamento a fasi, consentono l'inizializzazione della VM e l'esecuzione in sicurezza di tutte le operazioni offerte dai componenti.

2.5 CLI (Command Line Interface)

Installando la versione appropriata di WasmEdge sul nostro dispositivo, otterremo il file binario `wasmedge` insieme al suo alias `wasmedge run`. In generale, questa

installazione ci consentirà di eseguire i file .wasm nel nostro ambiente di runtime. Tuttavia, ci sono alcuni opzioni importanti da considerare.
Alcune delle opzioni più rilevanti includono (si può eseguire `wasmedge -h` per visualizzare l'elenco completo):

1. `--reactor modulo_da_caricare`

   In questo modo abbiamo pre-caricato il modulo in questione su wasmedge ed eseguendo

   ```
   wasmedge --reactor modulo_da_caricare funzione_modulo parametri_funzione
   ```

   potremo eseguire la funzione esportata dal suo modulo.

2. `--dir guest_path:host_path:(readonly|writeonly|readwrite)`

   Permette di mappare un file dal file system da cui stiamo lanciando il comando (guest) al file system interno alla vm che verrà eseguita da WasmEdge, il binding sarà reso possibile tramite WASI e infatti sarà possibile anche specificare i permessi che la vm avrà sul nostro file.

3. `--env NOME_VAR=VALORE_VAR`

   In questo modo possiamo impostare o creare delle variabili globali nella vm.

4. `--enable-all-statistics`

   Vengono elencate tutte le statistiche dell'esecuzione (durata, numero delle istruzioni, sforzo computazionale o gas). Sono poi presenti opzioni per ottenere le singole statistiche.

5. Limitazione delle risorse

   (a) `--time-limit MILLISECOND_TIME`

   Interrompe l'esecuzione dopo tot millisecondi.

(b) `--gas-limit GAS_LIMIT`

Interrompe l'esecuzione dopo tot unità di gas.

(c) `--memory-page-limit PAGE_COUNT`

Interrompe l'esecuzione dopo aver istanziato tot pagine (64 KB) in ogni istanza della memoria.

6. Proposte di esecuzioni alternative

Sono una serie di comandi per abilitare o disabilitare delle modalità di esecuzione specifiche, che vengono proposte di volta in volta.

7. Compilatore AOT

(a) `--dump`

Permette di generare il codice sorgente LLVM IR in due file:

- wasm.ll: conterrà il codice sorgente LLVM IR nel suo stato originale, senza ulteriori ottimizzazioni. Il codice sorgente può essere più leggibile ma potenzialmente meno efficiente in termini di esecuzione.
- wasm-opt.ll: conterrà il codice sorgente LLVM IR che è stato sottoposto a ottimizzazioni da parte del compilatore LLVM.

(b) `--interruptible`

Il .wasm generato supporterà l'esecuzione interrompibile (ovvero dove è possibile interrompere l'esecuzione di un programma WASM in modo controllato o di recuperare lo stato dell'esecuzione).

(c) `--generic-binary`

Genera un file binario generico permette che è indipendente dall'architettura della CPU sottostante. Il file sarà più grande e meno efficiente ma portabile.

(d) `--optimize LEVEL`

Scegliendo un livello di ottimizzazione tra [0,1,2,3,s,z] andremo a selezionare il livello di ottimizzazione della nostra LLVM. Nello specifico, i valori da 0 a 3 permettono una semplice ottimizzazione incrementale. I valori *s* e *z* indicano una ancora maggiore riduzione della dimensione.

2.6 Docker e WasmEdge

Docker[3] è un sistema per la creazione, la distribuzione e l'esecuzione di applicazioni all'interno di container. Egli permette di eseguire dei container con all'interno il WasmEdge runtime, consentendo quindi la loro esecuzione isolata, veloce, sicura e portatile ma permettendo anche una composizione efficace di più container per applicazioni più complesse e strutturate.

2.6.1 Struttura Docker

Per comprendere bene questo importante strumento, analizziamone l'architettura e il funzionamento in generale.

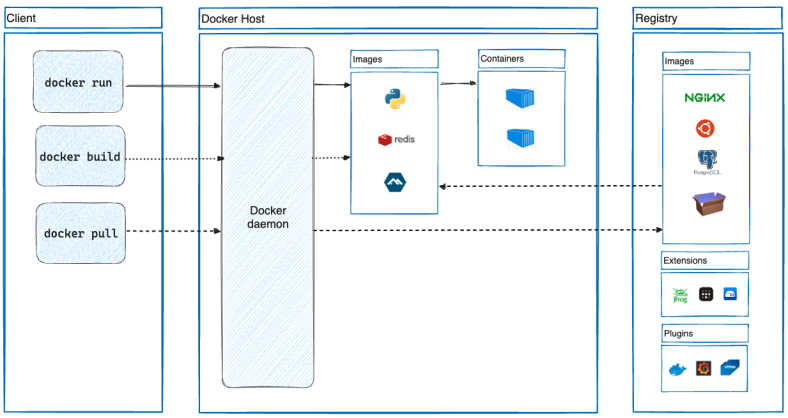

Figura 2.5: Archiettura generale di Docker [8]

[3] https://docs.docker.com

Docker è realizzato con un'architettura C/S e presenta tre componenti fondamentali:

1. **Client**

   Il client Docker consente agli utenti di formulare richieste per l'esecuzione, la creazione o il caricamento/scaricamento di immagini. Questa interfaccia utente fornisce un mezzo di comunicazione con il servizio Docker Engine, consentendo agli utenti di gestire le operazioni sui container in modo agevole.

2. **Docker Host** (Docker Engine)

   Il Docker Engine è il cuore del sistema Docker. Al suo interno ospita il processo Docker Daemon (dockerd), il quale rimane costantemente in esecuzione per accettare le richieste provenienti dal cliente e per elaborarle. Inoltre, il Docker Host mantiene un repository locale di immagini, che possono essere eseguite, create e scaricate/caricate dal/sul registro. In un'apposito repository di container, vengono eseguite le istanze delle immagini richieste dal cliente.

3. **Registry** (Docker Hub)

   Contiene tutte le immagini pubblicate dai vari host e le mantiene allo scopo di renderle disponibili in caso debbano venire scaricate.

Anche se non riguardano l'architettura generale, sono molto importanti anche container e immagini.

- **Immagine**

  Un'immagine Docker è un pacchetto leggero, autonomo ed eseguibile che include tutto il necessario per eseguire un'applicazione, compreso il codice, le librerie di sistema, le variabili d'ambiente e le dipendenze. La definizione di un'immagine Docker viene fornita tramite un file di configurazione chiamato Dockerfile. Questo file contiene una serie di istruzioni che vengono eseguite sequenzialmente da un 'builder' per costruire l'immagine desiderata.

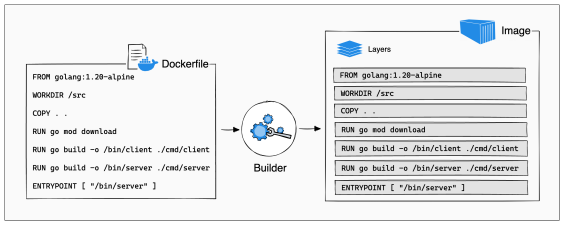

Figura 2.6: Creazione di un'immagine Docker, dato un Dockerfile [12]

- **Container**

  Un container è un'istanza eseguibile di un'immagine. Quando viene lanciato un container tramite un'immagine, si vanno a leggere i vari layer e tramite quei comandi viene fatto il provisioning del file system e del sistema operativo.

Per una migliore comprensione, si sottolinea che Docker implementa i propri servizi appoggiandosi a Containerd, un sistema di runtime per container che gestisce l'intero ciclo di vita del container.

2.6.2 Docker + Wasm

Recentemente, Docker ha introdotto il supporto per i container WASM. Containerd, per gestire il ciclo di vita di un container, usa un containerd-shim. Questo è un processo che si trova tra containerd e il runtime del container e permette al primo di eseguire operazioni sul secondo, in quanto ne astrae i dettagli tecnici. In sostanza quindi, per integrare qualsiasi runtime è sufficiente fornire un adeguato containerd-shim.

Infatti, per integrare WasmEdge è stato realizzato containerd-wasm-shim, che ci permette di usare l'architettura sandboxed dei moduli wasm al posto dei container, ma nella pratica rimanendo trasparenti per l'utente. Abilitando il supporto wasm quindi vengono scaricati tutti i containerd-shim per runtime WASM, tra cui quello per WasmEdge, ovvero `io.containerd.wasmedge.v1` .

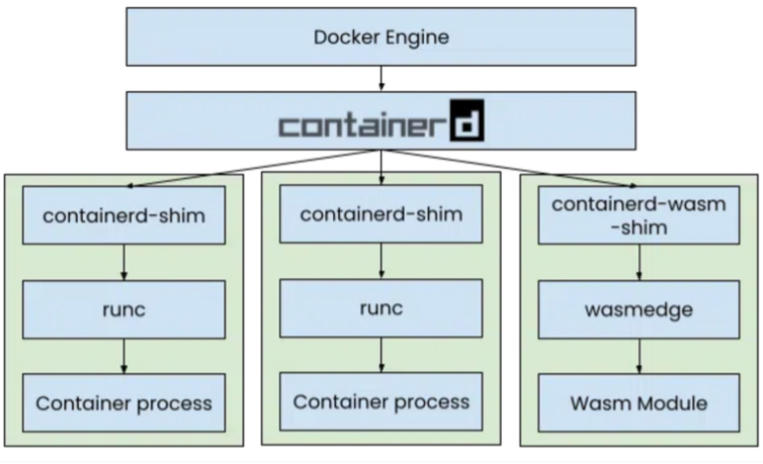

Figura 2.7: Differenza tra container classico e container WASM [10]

Inoltre, bisogna aggiungere che è necessario specificare la piattaforma di esecuzione, soprattutto se si esegue un runtime personalizzato, per assicurarsi che quest'ultimo sia compatibile con l'architettura di destinazione.
Ad esempio se vorrò eseguire un'immagine WASM dovrò usare:

```
docker run \
    --runtime=io.containerd.wasmedge.v1 \
    --platform=wasi/wasm \
    <nome_immagine>
```

2.6.3 Docker vs Wasm

Possiamo quindi mettere a confronto i container Docker (Linux) e i moduli WASM. Per quanto riguarda **le dimensioni e la velocità iniziale**[4], i moduli wasm sono 10 volte più leggeri e 10 volte più rapidi nella partenza rispetto ai container Linux classici, in quanto i primi non necessitano di includere e avviare librerie e servizi

[4] https://wasmedge.org/docs/start/build-and-run/docker_wasm

Linux.

Per quanto riguarda **l'isolamento e la sicurezza**, i container wasm limitano l'accesso alle risorse tramite un ambiente sandbox isolato tramite WASI, mentre i container Linux condividono il sistema operativo e questo potrebbe comportare una superficie d'attacco più ampia.

Per quanto riguarda **la portabilità**, i container wasm sono indipendenti dall'architettura sottostante mentre i container Linux dipendono appunto da architettura e sistema operativo sottostante.

Per quanto riguarda **la scalabilità**, vincono ancora i moduli wasm in quanto, essendo un modulo meno 'costoso' di un container, possono risultano più convenienti anche su larga scala.

Per quanto riguarda **la flessibilità**, i container Linux possono eseguire una qualsiasi applicazione supportata da un sistema operativo specifico, mentre i moduli wasm sono limitati alla compatibilità con i runtime Wasm disponibili e ai linguaggi che possono essere compilati in Wasm.

Per quanto riguarda **la facilità d'uso**, i container classici sono più avvantaggiati rispetto ai moduli wasm in quanto più diffusi e maturi. I primi dispongono di un ecosistema ricco di strumenti per la gestione (orchestratori come Kubernetes/-Docker e repository di immagini molto forniti come Docker Hub). Inoltre, i primi possono venire composti e automatizzati facilmente mentre i secondi hanno limitazioni di sicurezza molto elevate che possono ostacolare l'automatizzazione.

Infine, per quanto riguarda **la gestibilità**, in moduli wasm sono avvantaggiati in quanto la sicurezza che offrono semplifica la pianificazione necessaria. Invece nei container classici dobbiamo fare i conti con un'infrastruttura più complessa.

	NaCl	Application runtimes (eg Node & Python)	Docker-like container	WebAssembly
Performance	Great	Poor	OK	Great
Resource footprint	Great	Poor	Poor	Great
Isolation	Poor	OK	OK	Great
Safety	Poor	OK	OK	Great
Portability	Poor	Great	OK	Great
Security	Poor	OK	OK	Great
Language and framework choice	N/A	N/A	Great	OK
Ease of use	OK	Great	Great	OK
Manageability	Poor	Poor	Great	Great

Figura 2.8: Confronto tra container Docker, moduli WASM, esecuzione nativa e altri runtime
[5]

2.6.4 Wasm sarà il nuovo Docker?

Data la superiorità di WASM rispetto a Docker in quasi tutti i campi che abbiamo illustrato precedentemente, si potrebbe pesare che il primo sia destinato a sostituire il secondo. Invece, come suggerisce il fondatore di Docker in 2.9, potremmo pensare i due strumenti come complementari.

Indubbiamente, i moduli WASM presentano una serie di vantaggi rilevanti in termini di leggerezza, sicurezza e portabilità. Questi aspetti li rendono particolarmente adatti a scopi specifici, come l'esecuzione di funzioni o il potenziamento delle prestazioni delle applicazioni Web. Nel contesto di applicazioni che richiedono un'ottimizzazione delle risorse o un'architettura altamente sicura, i moduli WASM si distinguono come una scelta efficace.

D'altra parte, Docker rappresenta una tecnologia di containerizzazione estremamente versatile che offre una maggiore flessibilità nell'esecuzione di applicazioni complesse e nella gestione delle risorse di sistema. La sua capacità di incapsulare

ambienti completi in container agevola lo sviluppo, la distribuzione e la scalabilità delle applicazioni su diversi sistemi. Docker si rivela particolarmente utile per applicazioni complesse che richiedono una vasta gamma di dipendenze e servizi.
In definitiva, entrambe queste tecnologie hanno un ruolo essenziale nell'ecosistema di distribuzione delle applicazioni. La scelta tra i moduli WASM e Docker dipenderà dalle esigenze specifiche di ciascun progetto e delle applicazioni coinvolte.

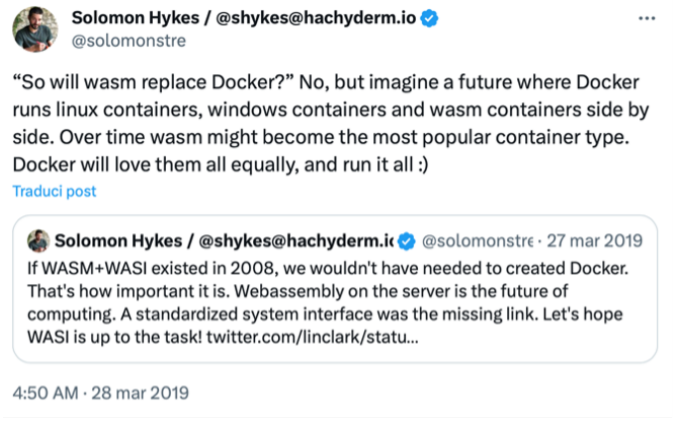

Figura 2.9: Tweet del fondatore di Docker, che esalta l'importanza di Wasm ma sottolinea l'opportunità di collaborare che hanno Wasm e Docker
[9]

2.7 Estensioni

WasmEdge non si limita solo all'implementazione delle estensioni ufficiali approvate da W3C, come ad esempio WASI. Riconosce che il processo di proposta e approvazione degli standard da parte di W3C può talvolta essere più lento rispetto alle necessità degli utilizzatori. Ciò è dovuto al fatto che la standardizzazione delle tecnologie punta alla garanzia della sicurezza e della robustezza di queste. Il divario temporale può comportare una mancanza di adattabilità e può richiedere l'implementazione di funzionalità non standard per soddisfare le esigenze immediate del mercato. Tuttavia, l'implementazione di funzionalità non standard può portare a

fragilità progettuali future.
WasmEdge ha affrontato questa sfida adottando una filosofia che gli consente di supportare estensioni non standard tramite l'uso di plug-in. I plug-in sono componenti software che estendono le funzionalità del runtime senza richiedere la ricompilazione dell'applicazione host. Questa scelta conferisce a WasmEdge il ruolo di un runtime sperimentale, che permette agli utenti di proporre e testare nuove idee e funzionalità. In seguito, queste proposte possono essere standardizzate o integrate in modo più robusto.
In questo modo, WasmEdge offre un ambiente flessibile in cui gli utenti possono sperimentare, sviluppare e testare nuove funzionalità senza dover aspettare i tempi di standardizzazione formale. Questo approccio rende WasmEdge un terreno fertile per l'innovazione e l'evoluzione delle capacità di runtime WebAssembly.

2.7.1 WasmEdge Plug-In System

WasmEdge ha introdotto una specifica architettura che agevola gli sviluppatori nell'estensione delle funzionalità attraverso l'uso di plug-in. Questa architettura consente ai programmatori di caricare e registrare funzioni tramite librerie condivise, offrendo un alto grado di personalizzazione del runtime per adattarsi alle necessità specifiche dell'applicazione. Grazie alla condivisione dei plug-in, è possibile progettare ulteriori funzionalità senza compromettere le prestazioni o la semplicità d'uso, e ciò contribuisce a migliorare la scalabilità delle funzioni offerte. Sono molti[5] i plug-in ufficiali di WasmEdge, alcuni dei quali sono stati brevemente affrontati tra le Host Function dei componenti WasmEdge e vanno dalla crittografia al machine learning.

[5] https://wasmedge.org/docs/contribute/plugin/intro#wasmedge-currently-released-plug-ins

2.8 Casi d'uso

Le prestazioni di WasmEdge lo rendono adatto a molti scenari, analizziamo ii principali.

2.8.1 SSR (Server Side Rendering)

Il Server Side Rendering è una tecnica di rendering ampiamente utilizzata nello sviluppo Web. Questa metodologia è finalizzata alla generazione di pagine Web dinamiche direttamente lato server e alla successiva trasmissione di queste pagine già complete al browser del cliente. L'elemento chiave di questa pratica è che, con l'SSR, il server invia al client il codice HTML finale, prontamente visualizzabile.
Questo approccio si contrappone al Client Side Rendering (CSR), in cui il server invia praticamente una struttura HTML vuota, e il rendering dell'intera pagina avviene nel browser del cliente mediante l'esecuzione di script JavaScript. Questi script sono, a loro volta, trasferiti dal server.
L'obiettivo primario dell'SSR è quello di migliorare l'esperienza dell'utente ottimizzando i tempi di caricamento delle pagine Web. Questo è possibile perché, con l'invio dell'HTML già generato dal server, il browser del cliente può renderizzare la pagina molto più rapidamente. Tuttavia, va notato che, se da un lato l'SSR può accelerare il caricamento, dall'altro potrebbe comportare una minore reattività, poiché gran parte del lavoro di rendering è già stato svolto lato server.
Qui WasmEdge risulta utile in quanto permette un'elaborazione complessa lato server e lo fa ad ottime prestazioni. Questo contribuisce in modo significativo all'accelerazione del processo di generazione dell'HTML, garantendo che la consegna della pagina avvenga nel minor tempo possibile. WasmEdge si integra con noti framework UI, come React, fornendo agli sviluppatori gli strumenti necessari per creare codice espressivo e potente, supportando al meglio la produttività nello sviluppo di interfacce utente dinamiche.

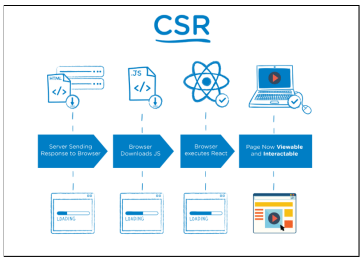

Figura 2.10: Client Side Render

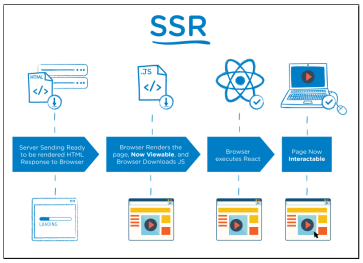

Figura 2.11: Server Side Render [11]

2.8.2 SaaS (Software as a Service)

Il modello di distribuzione del software basato su microservizi prevede l'hosting di applicazioni software su server remoti, rendendole accessibili agli utenti tramite Internet. Invece di installare il software sui dispositivi locali, gli utenti sfruttano le funzionalità del software direttamente attraverso un browser Web.
Per garantire la distribuzione efficiente di queste applicazioni, è essenziale utilizzare runtime leggeri, sicuri e performanti su cui eseguire i microservizi. Questi runtime costituiscono la base su cui vengono avviati e gestiti i componenti dell'applicazione, consentendo una distribuzione flessibile e scalabile delle funzionalità software.
Di seguito si riporta uno schema generico di un server cloud e dei servizi che offre e a che livello questi si trovano.

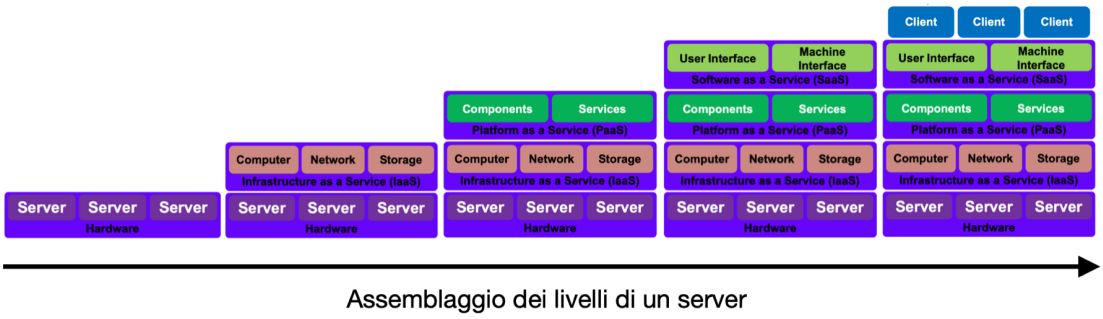

Figura 2.12: Struttura a livelli di un fornitore di servizi tra cui troviamo il SaaS [6]

Nella figura, alla base troviamo l'architettura reale (hardware) e i suoi prodotti software (server). A di sopra di quest'ultimo viene disposto un livello di infrastruttura (IaaS) che permette la distribuzione dei servizi cloud. Successivamente troviamo uno strato di piattaforma (PaaS) che fornisce servizi standard e componenti modulari fruibili da remoto agli strati superiori. I componenti modulari permettono poi di evitare la gestione dell'intero stack sistemistico, e si scrive la logica delle applicazioni. A concludere lo strato del server abbiamo lo strato software (SaaS) che mette a disposizione applicazioni preinstallate a cui fornire solamente configurazione e dati. Infine avremo i client che da remoto, connettendosi al server, possono usufruire dei vari servizi.

2.8.3 Serverless SaaS

Nel contesto dell'architettura serverless, le richieste degli utenti non vengono indirizzate a server permanenti, ma a server che attivano funzioni specifiche al momento della richiesta. WasmEdge svolge un ruolo fondamentale in questa dinamica, consentendo l'esecuzione di tali funzioni in un ambiente sicuro, flessibile, portabile e altamente scalabile, senza necessità di infrastrutture server sottostanti fisse.

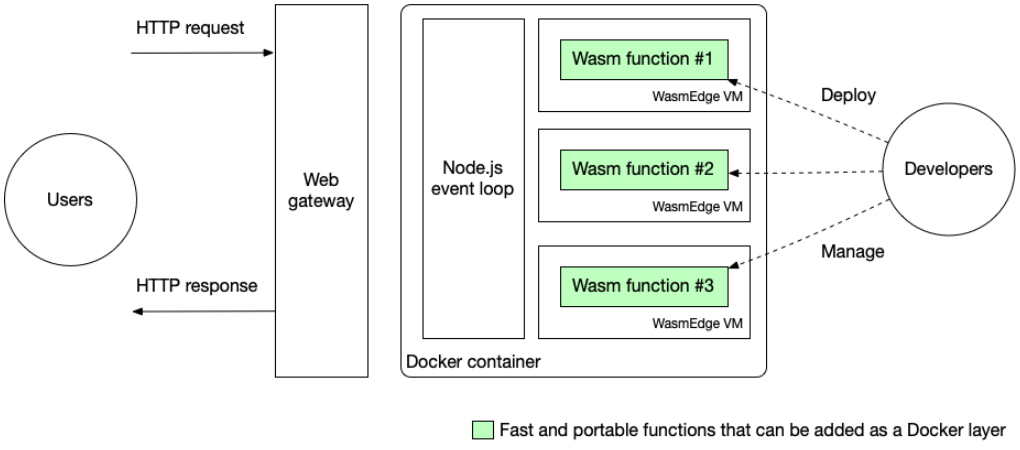

Figura 2.13: Architettura più specifica di un possibile Serverless SaaS [17]

Le funzioni serverless possono essere ospitate su diverse piattaforme, tra cui AWS Lambda, Netlify e altre[6]. Queste piattaforme di cloud computing forniscono ambienti serverless dove gli utenti, previa registrazione, possono caricare le proprie funzioni. Queste funzioni vengono eseguite in risposta a eventi specifici o chiamate, con la gestione automatica della scalabilità e dell'esecuzione delle funzioni chiamate.

Per avere un'idea più chiara dell'importanza di WasmEdge, prendiamo ad esempio AWS Lambda. Un recente sondaggio condotto da DataDog[7] ha rivelato che la maggior parte del codice eseguito su questa piattaforma era scritto in linguaggi di alto livello, come JavaScript e Python. Tuttavia, questi linguaggi possono essere notevolmente più lenti rispetto ai linguaggi di basso livello, come C o C++, soprattutto quando si tratta di funzioni computazionalmente intensive, come l'elaborazione di video, audio o immagini.

È interessante notare che WasmEdge esegue codice WebAssembly, che è un bytecode a basso livello, privo di queste limitazioni di velocità. Questa caratteristica è particolarmente evidente quando si confronta con gli svantaggi dei linguaggi di alto livello, come riportato anche dalla rivista Science[13], in cui si sottolinea che tali linguaggi possono essere fino a 60.000 volte più lenti, soprattutto quando utilizzati per funzioni particolarmente esigenti dal punto di vista computazionale.

2.8.4 Smart Device Apps

Le prestazioni di WasmEdge sono vantaggiose anche per le applicazioni da dispositivo, in quanto queste richiedono un'elevata interattività con l'interfaccia utente che, come abbiamo visto anche per il SSR, può venir implementato da un runtime con prestazioni particolarmente vantaggiose.

[6] https://wasmedge.org/docs/category/serverless-platforms/

[7] https://www.datadoghq.com/state-of-serverless

2.9 Competitor

Come visto in precedenza, sono tantissimi i runtime wasm presenti sul mercato. Ognuno di essi si distingue per le prestazioni particolarmente vantaggiose in un particolare caso d'uso o per le sue funzionalità. In questo contesto così fornito è difficile fare paragoni accurati e per questo considereremo solo altri 2 runtime come Wasmtime e Wasmer, questi infatti rappresentano un esempio di rigidità e di flessibilità nei confronti dello standard Wasm.

2.9.1 Wasmtime

Wasmtime[8] è il runtime ufficiale di WebAssembly sviluppato dalla Bytecode Alliance[9]. La caratteristica principale di Wasmtime è la sua ferma aderenza agli standard di WebAssembly, senza alcuna deviazione. Questo significa che Wasmtime segue scrupolosamente le specifiche di WebAssembly, garantendo un alto livello di conformità e compatibilità con il bytecode Wasm. Inoltre, Wasmtime si distingue per l'uso esclusivo del compilatore Cranelift[10], noto per le sue ottimizzazioni mirate al bytecode Wasm.
La rigidità di Wasmtime nel rispettare gli standard Wasm lo rende una scelta ideale in scenari in cui la sicurezza e la compatibilità sono prioritarie. Questo runtime è particolarmente adatto per applicazioni in cui non è possibile accettare compromessi sulla conformità ai standard e si richiede un elevato grado di affidabilità.

[8] https://docs.wasmtime.dev

[9] https://bytecodealliance.org

[10] https://github.com/bytecodealliance/wasmtime/tree/main/cranelift/docs/

2.9.2 Wasmer

Wasmer[11] è un altro runtime WebAssembly che si contraddistingue per la sua flessibilità. Wasmer implementa la compilazione del bytecode Wasm utilizzando una varietà di back-end, tra cui Cranelift, LLVM e SinglePass. Questa diversificazione offre agli sviluppatori una gamma di opzioni di compilazione da cui scegliere, consentendo loro di adattare il runtime alle specifiche esigenze del progetto.
La flessibilità di Wasmer lo rende particolarmente vantaggioso in situazioni in cui le prestazioni sono fondamentali e i compromessi sulla conformità possono essere accettati. La possibilità di selezionare il compilatore più adatto al proprio caso d'uso consente di ottimizzare le prestazioni in modo più specifico.

[11] https://docs.wasmer.io

Capitolo 3

Prototipo

Dopo un'analisi dettagliata di WasmEdge e una panoramica delle tecnologie circostanti, si procederà con la creazione di un prototipo. Il fine è quello di fornire uno strumento per stressare e testare efficacemente la tecnologia in diversi punti. In questo modo sarà possibile valutare i limiti, i vantaggi e la disponibilità della tecnologia allo stato dell'arte.

3.1 Descrizione dell'applicazione

Il prototipo realizzato consiste in un'applicazione da linea di comando progettata principalmente per mettere sotto pressione punti specifici del runtime che lo esegue, permettendo di personalizzare i parametri di stress tra vari livelli.
L'**obiettivo** dell'applicazione è simulare approssimativamente lo stress che potrebbe interessare una applicazione, fornendo un supporto nella fase di progettazione e scelta tecnologica. In questo contesto, l'applicazione fornisce dati utili per la selezione della tecnologia più adatta al progetto. I parametri di stress riguardano la lettura/scrittura di un file e l'utilizzo di algoritmi computazionalmente impegnativi.

L'applicazione esegue quindi 3 operazioni in sequenza:

1. Lettura di un file di testo
2. Riordinamento del testo
3. Scrittura del testo riordinato

3.1.1 Come si usa

Entrando nel dettaglio del suo utilizzo, l'applicazione fornisce una interfaccia da terminale del tipo:

```
<applicazione> carico algoritmo
```

Dove:

- `applicazione` : rappresenta il codice da eseguire e dipenderà dal runtime che vogliamo utilizzare.
- `carico` : rappresenta la fascia di carico in input. Può essere un numero da 1 a 7 che corrispondono rispettivamente a un file da 1KB, 10KB, 100KB, 1MB, 10MB, 100MB e 500MB.
- `algoritmo` : rappresenta il livello di complessità dell'elaborazione che vogliamo apportare al nostro contenuto in input. Può essere un numero da 1 a 3 che corrspondono a BubbleSort, MergeSort e QuickSort.

3.1.2 Come funziona

L'applicazione legge da una cartella predefinita un file specifico, precedentemente generato casualmente in base al carico richiesto. Il contenuto del file viene successivamente salvato nella memoria del processo tramite un array, che può consistere in vettori da centinaia di migliaia o addirittura milioni di caratteri.

Successivamente, a seconda del livello di elaborazione richiesto, vengono impiegati diversi algoritmi di ordinamento per ordinare il vettore memorizzato. È importante notare che la classificazione degli algoritmi basata su un solo parametro è una semplificazione.

BubbleSort

Il BubbleSort[1], anche se intuitivo e facile da implementare, si presenta come l'algoritmo meno efficiente e più lento, caratterizzato da una complessità temporale di $O(n^2)$ nel caso peggiore. Questo metodo di ordinamento opera confrontando coppie di elementi nell'array e spostando il più grande verso la fine. In particolare, il BubbleSort mostra le sue limitazioni quando si tratta di array di grandi dimensioni, in cui il numero di confronti e scambi aumenta esponenzialmente, risultando in una performance non ottimale.

QuickSort

Il QuickSort[2], invece, si distingue per la sua velocità in situazioni ottimali, con una complessità temporale di $\Omega(n \log_2 n)$. La sua strategia di suddividere l'array in base a un elemento pivot e successivamente ordinare le parti sinistra e destra rende il QuickSort particolarmente efficiente su array più piccoli o dataset ridotti. Tuttavia, è fondamentale considerare la sua complessità nel caso peggiore di $O(n^2)$, che si verifica quando la scelta del pivot è sfortunata.

[1] https://it.wikipedia.org/wiki/Bubble_sort

[2] https://it.wikipedia.org/wiki/Quicksort

MergeSort

Infine il MergeSort[3] offre una complessità temporale di $\Theta(n \log_2 n)$, mantenendo un rendimento costante e prevedibile indipendentemente dalle condizioni di input. Questa caratteristica rende il MergeSort una scelta affidabile e efficiente per array di dimensioni significative. Tuttavia, va notato che il MergeSort richiede uno spazio aggiuntivo nella memoria per gli array ausiliari utilizzati durante la procedura di fusione. Questo aspetto può influire sulla sua idoneità in scenari in cui la gestione efficiente della memoria è cruciale, come nel nostro caso.

Per realizzare quanto descritto finora insieme alla logica di testing, il **prototipo** è stato progettato come segue, i suoi componenti verranno spiegati in seguito:

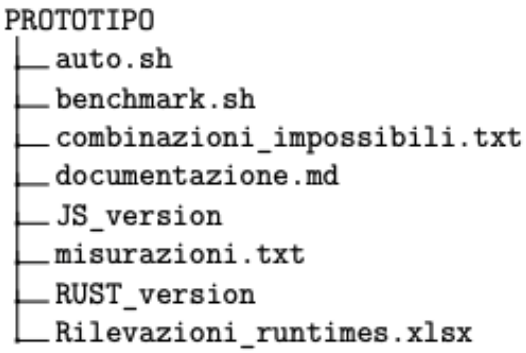

Figura 3.1: Struttura del prototipo

3.2 Implementazione

Il file di input è stato generato casualmente per evitare risultati anomali durante l'ordinamento. Per garantire la massima casualità nella selezione dei caratteri alfanumerici, abbiamo utilizzato come sorgente il dispositivo di sistema `/dev/urandom`,

[3] https://it.wikipedia.org/wiki/Merge_sort

implementato dal kernel per produrre uno stream pseudocasuale illimitato. Questo si basa su un pool di entropia in continuo aggiornamento, derivato da vari fattori come l'intervallo di pressione dei tasti, le interruzioni di sistema, il movimento del mouse, e così via. Il risultato ottenuto è stato successivamente filtrato per assicurare che la dimensione fosse esatta e che i caratteri fossero quelli richiesti. Questo è stato ottenuto tramite il seguente comando bash.

```
cat /dev/urandom | LC_CTYPE=C tr -dc '[:alnum:]' | head -c <num_byte>
```

La scrittura e la lettura del file avvengono mediante le tecnologie dei linguaggi scelti, argomenti che approfondiremo più avanti nel documento.
È cruciale notare che le dimensioni dei file sono state selezionate per fornire un'idea di un carico con crescita esponenziale, bilanciando questa scelta con la dimensione massima consentita da tutti i runtime. Ciò ha impedito l'utilizzo di carichi nell'ordine dei gigabyte.
Per quanto riguarda l'implementazione degli algoritmi di ordinamento, le versioni ricorsive di QuickSort e MergeSort sono conosciute per la loro efficienza, ma richiedono un notevole utilizzo dello stack a causa delle chiamate ricorsive. Tuttavia, quando si lavora con carichi anche solo medi, si può facilmente incorrere in un errore di stack-overflow che impedisce il completamento del test. Di conseguenza, gli algoritmi sono stati riscritti in versione iterativa al fine di consentire tutte le misurazioni necessarie.

3.2.1 JavaScript

Il prototipo è stato prima sviluppato in JavaScript nella cartella 'JS_version' ed eseguito con Node.js, usando i seguenti moduli.

```
const fs = require('fs');
const { bubbleSort, mergeSort, quickSort } = require('./mioSorting');
```

La lettura e la scrittura di file sono state implementate utilizzando il modulo `fs`, il quale gestisce la memorizzazione del contenuto in una stringa e la sua successiva scrittura. La stringa viene successivamente convertita in un array di caratteri al fine di agevolarne la manipolazione 'in-place' da parte degli algoritmi di ordinamento. L'ordinamento è stato implementato con la libreria `mioSorting.js`, la quale espone le funzioni necessarie per ordinare il vettore di caratteri.

```
1  // Scelta del file in base alla complessita
2  var path;
3  switch(carico){
4      case "1" : path = "./files_input/1KB.txt"; break;
5      case "2" : path = "./files_input/10KB.txt"; break;
6      case "3" : path = "./files_input/100KB.txt"; break;
7      case "4" : path = "./files_input/1MB.txt"; break;
8      case "5" : path = "./files_input/10MB.txt"; break;
9      case "6" : path = "./files_input/100MB.txt"; break;
10     case "7" : path = "./files_input/500MB.txt"; break
11 }
12
13 // Lettura
14 let contenuto = [];
15 readFile(path, 'utf8', async (err, data) => {
16     if (err) {
17         console.error('Errore durante la lettura del file:', err);
18         return;
19     }
20
21     // Da stringa ad array
22     contenuto = data.split('');
23
24     // Elaboro il contenuto
25     switch(algoritmo){
26         case "1" : bubbleSort(contenuto); break;
```

```
27          case "2" : mergeSort(contenuto); break;
28          case "3" : quickSort(contenuto); break;
29      }
30
31      path = path.replace("input", "output");
32      var testo = contenuto.toString().replace(/,/g, "");
33
34      // Scrittura
35      writeFile(path, testo, 'utf8', (err) => {
36          if (err) {
37              console.error('Errore durante la scrittura del file:',
                    err);
38              return;
39          }
40      });
41  });
```

Listato 3.1: Versione JavaScript del prototipo

3.2.2 Rust

Rust[4] è un linguaggio di programmazione avanzato che si distingue per il suo sistema di ownership, il quale consente una gestione efficiente della memoria senza ricorrere a un garbage collector. La tipizzazione statica e il sistema di inferenza dei tipi permettono a Rust di offrire un elevato livello di sicurezza a livello di compilazione, minimizzando gli errori durante l'esecuzione.

Il sistema di ownership di Rust consente un controllo preciso sulla mutabilità dei dati e previene problemi comuni come le race condition e gli accessi concorrenti non sicuri. In aggiunta, Rust implementa il concetto di "capabilities" attraverso la gestione di riferimenti, permettendo ai programmatori di definire con precisione

[4] https://doc.rust-lang.org/book/

chi può accedere e modificare i dati, contribuendo così a evitare errori di accesso indesiderati.

Cargo, il gestore di pacchetti e sistema di build di Rust, semplifica la gestione delle dipendenze, la creazione di nuovi progetti e la compilazione del codice. Integra un sistema di test e offre un ambiente completo per lo sviluppo di progetti Rust.
Per creare un nuovo progetto Rust è pertanto necessario fare uso di Cargo.

```
cargo new RUST_version
```

In questo modo genereremo la cartella del progetto. Notiamo un file `Cargo.toml`, in cui saranno contenute le librerie esterne che vorremo usare e una cartella `src`, in cui andrà inserito il codice sorgente da compilare. La cartella che verrà generata ha la seguente struttura :

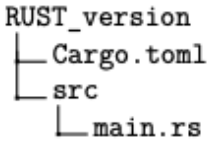

Figura 3.2: Struttura generica di un progetto Rust

Quando si compila un programma Rust, è possibile specificare il target di destinazione, ovvero un sistema operativo specifico o un'architettura di un processore per il quale si desidera generare l'eseguibile. Verrà preparato quindi l'ambiente Rust per compilare il codice sorgente .rs in un formato WebAssembly compatibile con WASI, così.

```
rustup target add wasm32-wasi
```

A questo punto basta scrivere il codice di nostro interesse in main.rs e creeremo l'eseguibile con

```
cargo build --target wasm32-wasi --release
```

In questo modo è stato realizzato l'eseguibile relativo al progetto, situato in `RUST_version/target/wasm32-wasi/prototipo.wasm`

Il prototipo è stato poi sviluppato in Rust nella cartella 'RUST_version', tramite i seguenti moduli, al fine di avere le stesse funzionalità viste precedentemente per poter venire poi compilato in wasm ed eseguito su WasmEdge, Wasmtime e Wasmer.

```
use std::env;
use std::fs::File;
use std::io::{Write, Read};
use mio_sorting::{bubble_sort, merge_sort, quick_sort};
```

```
1  // Scelfo il file in base alla complessita
2  let mut input_file_name = "";
3  match carico {
4      1 => {input_file_name = "files_input/1KB.txt";}
5      2 => {input_file_name = "files_input/10KB.txt";}
6      3 => {input_file_name = "files_input/100KB.txt";}
7      4 => {input_file_name = "files_input/1MB.txt";}
8      5 => {input_file_name = "files_input/10MB.txt";}
9      6 => {input_file_name = "files_input/100MB.txt";}
10     7 => {input_file_name = "files_input/500MB.txt";}
11     _ => {println!("Carico non gestito");}
12 }
13
14 // Leggo il contenuto
15 let mut content: Vec<char> = read_file(input_file_name).chars().
       collect();
16
17 // Riordino il contenuto
18 match algoritmo {
19     1 => {bubble_sort(&mut content);}
```

```
20      2 => {merge_sort(&mut content);}
21      3 => {quick_sort(&mut content);}
22      _ => {println!("Algoritmo non gestito");}
23  }
24
25  // Scrivo il contenuto
26  let output_file_name = input_file_name.replace("input", "output");
27  create_file(&output_file_name, &content.iter().collect::<String>())
        ;
28  }
```

Listato 3.2: Versione Rust del prototipo

3.3 Metodologia di test

In JavaScript, sarebbe stato possibile implementare strutture di benchmark per valutare con maggiore precisione lo stress del sistema. Queste strutture avrebbero consentito di misurare attentamente lo stato del sistema prima e dopo l'esecuzione di operazioni "stressanti" da parte dell'applicazione. Tuttavia, WASI, per questioni di sicurezza, non permette l'accesso ai dati del sistema da parte del processo, rendendo impossibile eseguire misurazioni e calcoli mirati.
Per cui, al fine di effettuare la misurazione, si è optato per ottenere le informazioni di sistema tramite uno script bash denominato 'benchmark.sh'. Questo script viene lanciato nel seguente modo:

```
./benchmark.sh <comando>
```

Esso esegue il comando (con i relativi parametri) e durante l'esecuzione monitora costantemente l'utilizzo della memoria da parte del processo e la sua occupazione della CPU. Le medie di questi valori vengono calcolate durante l'intera esecuzione e restituite insieme alla durata totale dell'esecuzione, ottenuta alla sua terminazione.

Tali parametri ci consentono di determinare approssimativamente l'impatto della nostra applicazione sul sistema. Nello specifico andremo a misurare:

- **Latenza (ms)**

  La latenza di un'operazione si definisce come il tempo che intercorre tra l'inizio e la fine dell'operazione stessa.

  $$L_{\text{op}} = \Delta t(\text{op}) = t(\text{op}_f) - t(\text{op}_i) \tag{3.1}$$

  Indica quindi la velocità con cui risponde la nostra applicazione ed è uno dei parametri più importanti nei servizi Web. Per misurarla si è adoperato il comando Unix `time` che effettua in automatico queste misurazioni.

- **Uso della CPU (%)**

  L'uso della CPU rappresenta la percentuale di tempo in cui il processore è dedicato all'esecuzione di istruzioni per un particolare processo o per l'intero sistema. Attraverso questo parametro si può rilevare quanta capacità di elaborazione non viene usata, rimanendo disponibile per altri processi. Per calcolare questo valore, si considera il periodo di tempo in cui la CPU è stata 'idle' o inattiva in due momenti, per poi ottenere il valore complementare attraverso la seguente formula: L'interpretazione del suo valore dipende quasi totalmente dal ruolo che il processo ha per il sistema. Risulta positivo che un processo critico abbia a disposizione tutta la CPU, mentre se ciò accade con un processo non importante diventa un problema, poiché vengono tolte risorse importanti a chi ne ha bisogno. Nella nostra misurazione, l'utilizzo percentuale della CPU è stato ottenuto tramite il comando Unix `time`.

- **Uso della Memoria in KB:**

  L'uso della memoria, rappresentato dal Resident Set Size (RSS), indica la quantità di RAM allocata al processo in un determinato momento. RSS include sia la memoria condivisa (come librerie in comune) che la memoria privata

del processo. Per misurare l'uso della memoria durante l'esecuzione di un'operazione, sono stati effettuati campionamenti periodici dei valori di RSS utilizzando il comando Unix `ps`. La media di questi valori campionati è stata quindi calcolata per ottenere l'uso medio della memoria durante l'intera operazione. In questo modo non viene considerata solo quanta memoria viene associata al processo ma anche per quanto tempo mantiene quelle risorse in media. La formula utilizzata è la seguente:

$$\mathrm{MEM}_{\mathrm{op}} = \frac{\sum \mathrm{RSS}_{\mathrm{op}}}{N} \tag{3.2}$$

Dove $\sum \mathrm{RSS}_{\mathrm{op}}$ rappresenta la somma dei valori campionati di RSS durante l'operazione ed N è il numero totale di campioni durante l'operazione.

3.4 Setup sperimentale

Per eseguire i test presentati è stato utilizzato un laptop con processore Intel Core i7 quad-core (1,2 GHz), 16 GB di Ram e sistema operativo MacOS Ventura 13.2.1. Per quanto riguarda i software utilizzati, sono state usate le seguenti versioni:

- Node.js v18.12.1
- Rust 1.73.0
- WasmEdge 0.13.4
- Wasmtime 14.0.4
- Wasmer 4.2.3

3.5 Risultati

Per ottenere stime accurate dei risultati, è fondamentale far eseguire le stesse combinazioni di esecuzioni ai runtime varie volte, calcolandone le medie. È da notare

che, per i prototipi, sono disponibili 7 file di input e 3 modalità di elaborazione, generando un totale di 21 combinazioni di esecuzione. Tuttavia, molte di queste configurazioni risultano prive di interesse pratico o sono fisicamente irrealizzabili (come riordinare milioni di elementi con BubbleSort).
Per gestire queste sfide in modo efficiente, è stato sviluppato uno script bash denominato 'auto.sh'. Questo script, mediante l'input di un numero N, esegue automaticamente tutte le combinazioni di comandi N volte, calcolandone le medie. I risultati vengono registrati nel file `misurazioni.txt`, che contiene le medie dei dati di tutte le combinazioni per ogni runtime.
Al fine di evitare l'esecuzione di combinazioni non praticabili, è stato creato il file `combinazioni_impossibili.txt`, il quale elenca le combinazioni da escludere dall'esecuzione di 'auto.sh'. I risultati sono stati successivamente organizzati nel foglio di calcolo `Rilevazioni_runtimes.xlsx`. Qui, sono stati raggruppati e classificati in base ai parametri di stress per produrre insiemi di dati rappresentabili e confrontabili.
Va sottolineato che i risultati sono stati ottenuti effettuando una media di 15 esecuzioni per combinazione tramite il comando `./auto.sh 15`.

3.5.1 Confronto generale

Confrontiamo ora i risultati ottenuti tra tutti i runtime. È stato anche precompilato il file .wasm utilizzando `wasmedgec` per valutare le capacità di questa potente opzione di WasmEdge, evidenziata nei grafici come 'WasmEdge AOT'. Tra tutte le combinazioni possibili, solo 9 sono risultate applicabili a tutti i runtime, consentendo un confronto completo tra di essi. Nonostante le potenzialità degli strumenti in esame, il BubbleSort si è rivelato così inefficiente da richiedere un utilizzo della CPU molto elevato per un lungo periodo, determinando la terminazione preventiva del processo da parte del sistema per motivi di sicurezza. Questa situazione ha reso impraticabile il test dell'algoritmo su Node e su alcuni runtime Wasm con carichi

elevati. Inoltre, Node.js ha manifestato notevoli difficoltà nell'esecuzione dell'applicazione oltre il carico da 1MB, rendendo arduo rappresentare le differenze con gli altri motori di esecuzione.

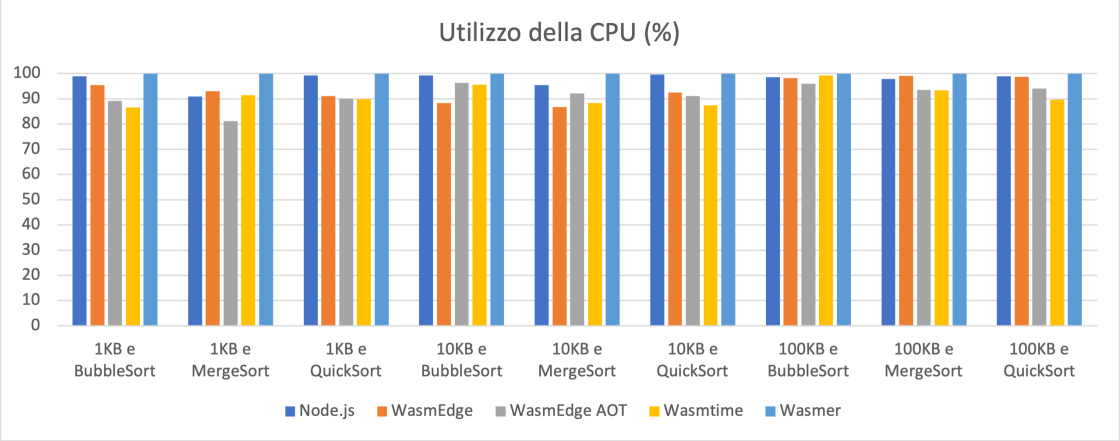

Figura 3.3: Utilizzo della CPU tra i diversi runtime

Dall'analisi dell'utilizzo della CPU (figura 3.3), emerge che tutti i runtime presentano valori significativamente elevati in qualsiasi condizione.
Si evidenzia, tuttavia, la tendenza di Wasmtime e WasmEdge AOT a utilizzare la CPU in misura inferiore rispetto agli altri runtime. Al contrario, Node.js e Wasmer hanno mostrato una tendenza a occupare quasi sempre il 100% della CPU. Questo è dovuto in gran parte alla numerosa quantità di operazioni richieste dagli algoritmi di sorting durante il confronto degli elementi degli array al fine di determinarne l'ordinamento.

È importante sottolineare come i grafici seguenti siano stati tagliati di diversi ordini di grandezza, in quanto l'uso di memoria e la latenza introdotta dal BubbleSort rendevano insignificanti e illeggibili gli altri valori.

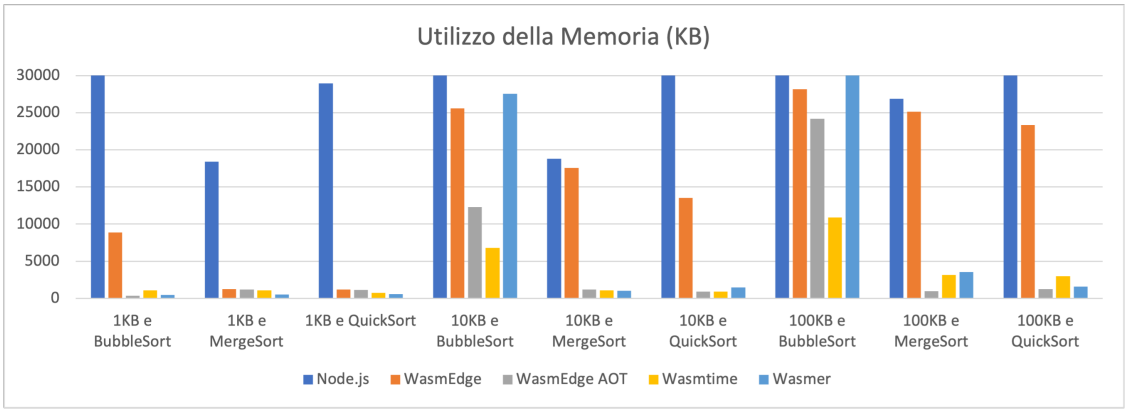

Figura 3.4: Utilizzo della Memoria tra i diversi runtime

Per quanto riguarda l'utilizzo della memoria (figura 3.4) invece si evince l'inefficienza di Node.js rispetto alle sue controparti di wasm. Infatti mentre il file .wasm contiene il codice essenziale per la sua esecuzione e non grava sul runtime, i file .js richiamano librerie JavaScript molto pesanti e Node.js come motore richiede molta memoria[5] per il suo funzionamento.

Ponendo l'attenzione su WasmEdge e i suoi pari, notiamo che il primo risulta spesso molto più inefficiente degli altri e ricalca l'andamento di Node.js. Se però consideriamo lo stesso runtime ma con un file .wasm compilato Ahead-Of-Time, il confronto si fa più interessante e WasmEdge risulta spesso la scelta più efficiente.

[5] https://www.ness.com/understand-how-to-reduce-memory-usage-of-promises-in-node-js

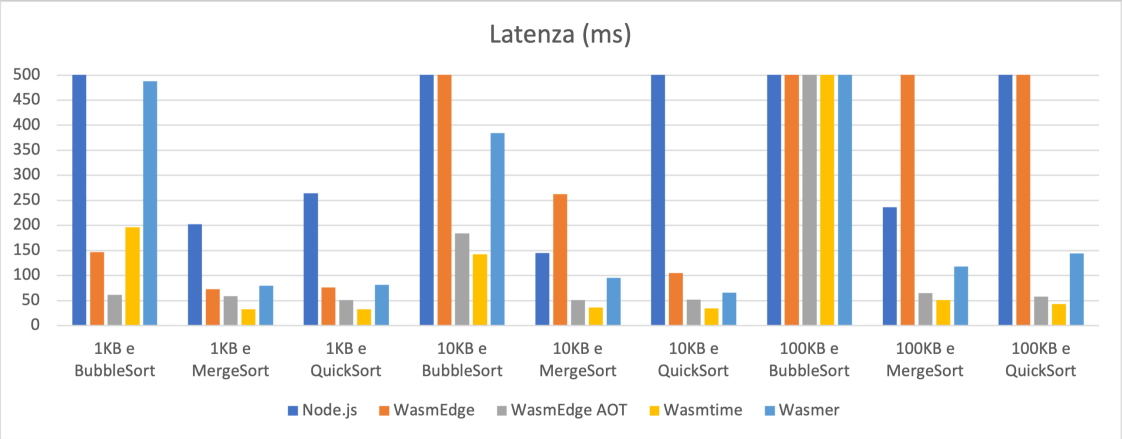

Figura 3.5: Latenze tra i diversi runtime

Osservando le latenze (figura 3.5), anche qui un andamento simile a quello osservato per l'utilizzo della memoria. Notiamo che WasmEdge AOT e Wasmtime risultano sempre i più veloci, presentando una latenza praticamente costante all'aumentare dell'ordine di grandezza del carico. Si evidenzia un problema che si dovrà tenere a mente anche nelle esecuzioni successive, ovvero la latenza e l'uso di memoria introdotte dal BubbleSort, che già con 100.000 elementi da ordinare, si distingue per avere ordini di grandezza molto più elevati degli altri.

3.5.2 Runtime wasm

Rispetto a Node.js, che perdeva troppa efficienza dopo i carichi medio-bassi, i runtime wasm hanno permesso l'esecuzione anche per carichi medio-alti. Come anticipato precedentemente, il BubbleSort impiegava troppo tempo per eseguire operazioni con così tanti elementi e la sua complessità temporale quadratica lo ha reso inutilizzabile per carichi con ordini di grandezza ancora superiori e quindi non verrà riportato.

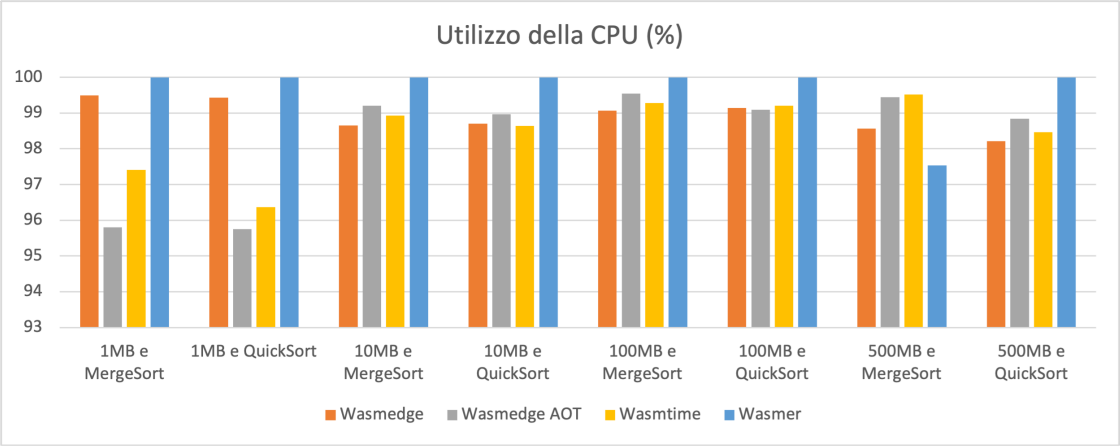

Figura 3.6: Utilizzo della CPU tra i runtime WASM

Osservando l'utilizzo della CPU (figura 3.6), si è notata ancora la tendenza di tutti i motori a mantenere valori molto alti, con pochissime differenze che riconfermano l'andamento osservato anche nel precedente confronto anche con Node.js.

I seguenti grafici, riguardo l'uso della memoria e riguardo le latenze, sono stati divisi in due parti a causa della grande differenza di scala introdotta dal carico esponenzialmente maggiore.

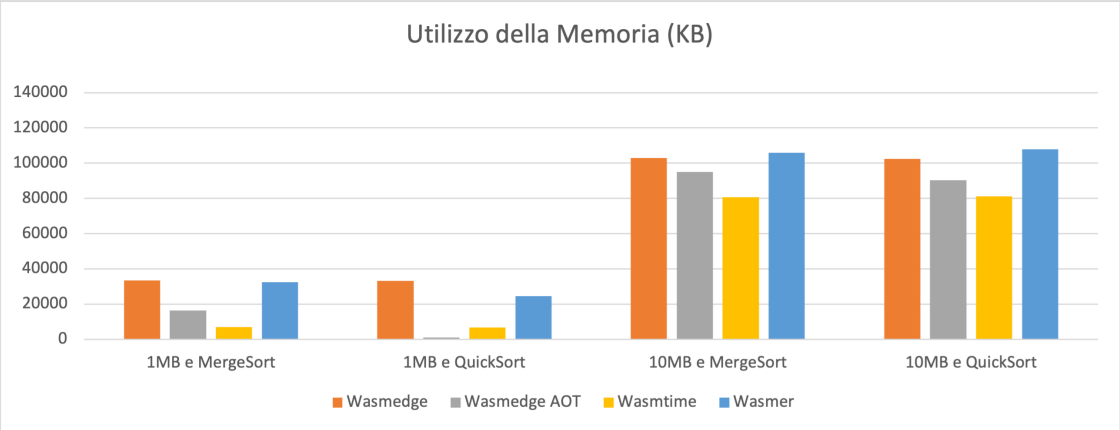

Figura 3.7: Utilizzo della Memoria tra i runtime WASM, fino a 10MB

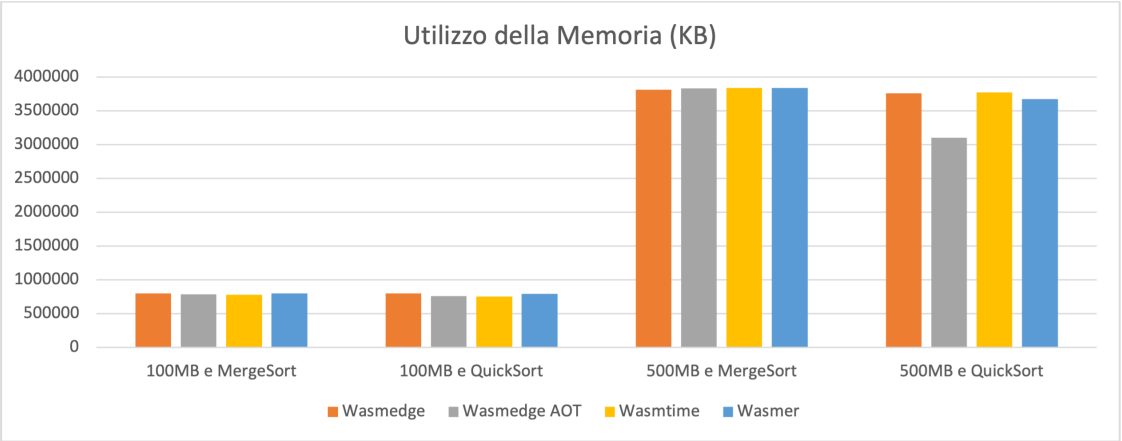

Figura 3.8: Utilizzo della Memoria tra i runtime WASM, per 100MB e 500MB

Aumentando notevolmente i carichi in ingresso possiamo notare come la memoria usata sia praticamente la stessa tra i runtime. Tranne per una particolare efficienza di WasmEdge AOT e Wasmtime per carichi di 1MB (figura 3.7). Si noti comunque come la memoria richiesta sia comunque abbastanza superiore al carico in ingresso, dovuto alle strutture dati usate dagli algoritmi di sorting e dagli array utilizzati in scrittura e lettura che si sommano.

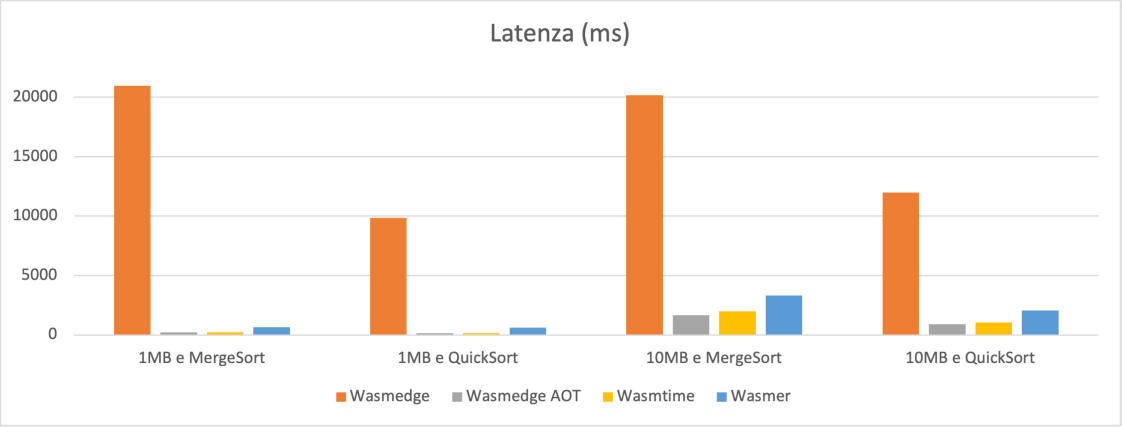

Figura 3.9: Latenze tra i runtime WASM, fino a 10MB

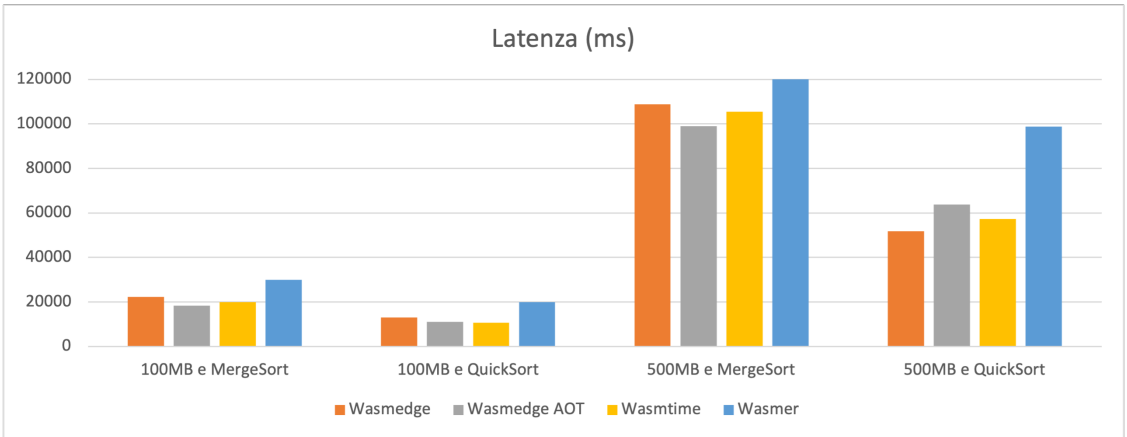

Figura 3.10: Latenze tra i runtime WASM, per 100MB e 500MB

Infine, osservando le latenze misurate con carichi medio-alti (figura 3.9), possiamo notare una grande lentezza di Wasmedge fino a 10MB rispetto agli altri competitor, che però si riprende e diventa molto più competitivo con carichi alti fino a 500MB (figura 3.10). Risultano però molto più veloci in media WasmEdge AOT e Wasmtime, soprattutto per i carichi medio-alti, mentre con carichi alti i motori tendono ad avere le stesse latenze.

3.5.3 Concorrenza

Inizialmente, il prototipo è stato progettato con l'obiettivo di avere un'estensione che consentisse anche la generazione di thread, al fine di avvicinarsi ulteriormente al comportamento di un'applicazione comune. Approfondendo il tema, ci si è tuttavia accorti di un ostacolo che ha comportato la modifica del progetto, e che è importante trattare alla luce degli sviluppi futuri che comporterà.
Nel corso dell'analisi e dei test sulle tecnologie, è emerso che Rust supporta il multithreading attraverso `std::thread`, mentre WASI non ha ancora implementato uno standard, rendendo di conseguenza impossibile il supporto all'operazione nei

runtime wasm. Tuttavia, è fondamentale notare che i thread, utilizzati per la gestione concorrente delle risorse, rappresentano già uno standard implementato nei browser e in altri runtime.

Thread in JavaScript (Node.js)

JavaScript supporta la concorrenza tramite il modulo **worker_threads**. Questo implementa il threading tramite un oggetto **Worker** che esegue in uno spazio di lavoro separato, consentendo l'esecuzione di codice in modo asincrono e parallelo rispetto al thread principale. Per comprendere meglio il funzionamento di questo modulo in Node.js, possiamo esaminare alcuni dettagli di implementazione.
Quando si crea un nuovo Worker, Node.js avvia un nuovo processo separato o un thread, a seconda delle configurazioni del sistema e dell'architettura. Questo nuovo thread esegue il codice specificato nel file fornito come argomento al costruttore del Worker.

```
const worker = new Worker('./codice_worker.js');
```

La comunicazione tra il thread principale e i Worker avviene attraverso un meccanismo di messaggistica basato su eventi. Il thread principale e i Worker possono scambiarsi messaggi utilizzando il metodo `postMessage`. La messaggistica è asincrona e consente la comunicazione senza dover condividere direttamente lo stato o le variabili, riducendo così i problemi di concorrenza. Inoltre, i Worker possono accedere a un contesto di esecuzione isolato, riducendo il rischio di condivisione accidentale di dati mutabili tra i thread e semplificando la gestione della concorrenza. L'implementazione di `worker_threads` sfrutta le capacità multithreading del sistema operativo per migliorare le prestazioni dell'applicazione, consentendo al programma di sfruttare i vantaggi delle architetture multicore, perché appunto ciascun worker thread può essere assegnato a un core di CPU diverso dal thread principale.

Thread in Wasm (Browser)

Nonostante al momento non sia possibile sfruttare la concorrenza nei runtime WebAssembly che abbiamo studiato, questa funzionalità è già stata implementata nei runtime wasm dei browser[7][**article:wasmThread2**]. WebAssembly presenta certamente una sfida significativa per il multithreading, soprattutto per quanto riguarda la memoria condivisa. Infatti, wasm utilizza una memoria "sandboxed", e la sua gestione non è banale. Come discusso nei capitoli precedenti, la memoria di un modulo wasm è rappresentata da un `ArrayBuffer` in JavaScript che funge da 'glue-code'. `SharedArrayBuffer` è una struttura simile già presente nella condivisione di memoria tra i Worker e il processo padre in JavaScript. Quindi i thread WebAssembly non sono una funzionalità separata, ma una combinazione di diversi componenti che consente alle app WebAssembly di utilizzare i tradizionali paradigmi multithreading sul Web. L'idea è infatti quella di modificare l'array che simula la memoria del modulo wasm (`WebAssembly.Memory`) per cambiare le proprietà della memoria stessa e renderla condivisibile tra diversi moduli (`WebAssembly.Module`) per permettere la concorrenza di questi ultimi.
Affinchè questo sia realizzabile è necessario l'uso del Worker JS. I thread WebAssembly utilizzano il costruttore `new Worker` per creare nuovi thread sottostanti. Ogni thread utilizza il metodo `Worker.postMessage()` per condividere il modulo wasm compilato e una memoria condivisa con gli altri thread. Questo stabilisce la comunicazione e consente a tutti i thread di eseguire lo stesso codice WebAssembly sulla stessa memoria condivisa senza dover passare da JavaScript.
Però sorge un problema fondamentale quando si parla di concorrenza su risorse condivise, ovvero la sincronizzazione degli eventi, per evitare che un thread intervenga su una risorsa che sta utilizzando un secondo thread, generando malfunzionamenti. Per superare anche questo ostacolo sono state quindi introdotte le istruzioni WebAssembly per l'accesso atomico alla memoria come `wait()` e `notify()`. Queste consentono a un thread di entrare in modalità di sospensione su un determinato

indirizzo in una memoria condivisa fino a quando un altro thread non lo riattiva.

Il threading, quando applicato a wasm, permette di ottenere risultati molto validi. Ad esempio[14], se consideriamo un'applicazione che effettua il rendering di un'immagine su Google Earth, possiamo constare visivamente la differenza tra un approccio single-threaded e uno concorrente. Il secondo infatti sarà in grado di produrre più frame al secondo e contemporaneamente di disperderne di meno rispetto al primo. La concorrenza di wasm sui browser ci mostra quindi il potenziale ancora nascosto fuori dal browser.

3.6 Sviluppi futuri

Dinanzi alla mancanza di uno standard per la gestione della concorrenza, la Bytecode Alliance ha recentemente avviato un processo[6] di standardizzazione per affrontare il problema. Il primo risultato è una proposta nella fase 1: `wasi-threads`[7]. Questa proposta mira a fornire un'API standard per la creazione di thread, operando a livello WASI e integrandosi con la proposta di threads a livello WebAssembly. La proposta a livello WebAssembly fornisce i blocchi fondamentali necessari per la gestione di memoria condivisa, operazioni atomiche e attese/notifiche. Il pseudo-codice della proposta sarebbe:

```
status wasi_thread_spawn(thread_start_arg* start_arg);
```

La funzione riceve in ingresso un puntatore a una struttura che contiene i dati necessari per avviare il nuovo thread. Successivamente prova a crearlo ed avviarlo, per poi restituire un numero intero univoco non negativo rappresentante l'ID del nuovo thread (TID) in caso di successo. Se si verifica un errore durante la generazione del

[6] https://github.com/WebAssembly/meetings/blob/main/process/phases.md

[7] https://bytecodealliance.org/articles/wasi-threads

thread, il valore sarà un numero negativo che indica il tipo specifico di errore.
Durante l'esecuzione sarà presente un'altra funzione (`wasi_thread_start`) che risiederà nell'istanza del modulo WebAssembly associata a ciascun thread generato utilizzando l'API stessa. Ovvero ogni volta che viene creato un nuovo thread, viene istanziato un nuovo modulo WebAssembly per quel thread, e la funzione `wasi_thread_start` è una funzione esportata all'interno di questo modulo. Il suo compito sarà quella di prendere il TID del thread e l'argomento di avvio per eseguire qualsiasi configurazione iniziale necessaria per il thread. Poi attiverà le sue funzionalità, definite dall'utente in `start_arg`. Infine alla terminazione della sua esecuzione, ne gestirà la terminazione e notificherà al thread principale l'avvenimento.
È importante ricordare che l'invocazione di `wasi_thread_spawn` dovrà essere seguita da WASI, che dovrà istanziare nuovamente il modulo, importarvici tutti gli stessi oggetti WebAssembly, calcolare un TID univoco e chiamare la funzione di ingresso esportata dell'istanza figlia con l'ID del thread e l'argomento di avvio.

Come già anticipato, la proposta è alla fase 1 (Feature Proposal). Le fasi successive prevedono la scrittura di un testo di specifica (Feature Description Available) su come implementare la proposta, per poi passare alla fase di implementazione (Implementation Phase) che produrrà un prototipo. Quest'ultimo poi sarà sottoposto ai test e alle revisioni della community durante la fase di standardizzazione (Standardize the Feature) e una volta approvata sarà disponibile agli sviluppatori (The Feature is Standardized).

Conclusioni

Nel contesto attuale di necessità di trovare alternative efficienti, veloci, scalabili e robuste alle attuali tecnologie per l'erogazione di servizi. Il nostro progetto di tesi ha evidenziato diversi punti di forza e alcune debolezze nell'utilizzo di WasmEdge. Inizialmente, è stato sottolineato che WasmEdge si presenta come un runtime ben strutturato, integrato nei container e in continua evoluzione, con la flessibilità di accogliere proposte non ancora standardizzate attraverso plug-in o estensioni, pur adottando lo standard WASI. Questo è utile per garantire sicurezza e solidità anche se ciò comporta una certa rigidità durante l'implementazione, come dimostrato dai limiti riscontrati nella concorrenza durante lo sviluppo del prototipo o dalla severa gestione degli accessi al file system.
Ciò ci ha però mostrato come gli sviluppatori siano continuamente all'opera per oltrepassare gli ostacoli che ora si pongono davanti allo strumento, e che apre ottime prospettive per gli sviluppi futuri di quest'ultimo.

Il confronto con Node.js ha confermato le aspettative. WasmEdge si è dimostrato spesso molto più veloce e ha anche gravato meno sulla memoria, mostrando un distacco elevato soprattutto per carichi minori.
Il confronto con gli altri runtime Wasm come Wasmtime e Wasmer ha evidenziato che WasmEdge ha ancora molti aspetti da perfezionare per poter essere altamente competitivo in ogni situazione, come spesso ha mostrato Wasmtime. Nonostante

ciò si è comunque dimostrato capace di prestazioni ottime, specialmente in termini di velocità con carichi sia elevati che ridotti. Nonostante risulti attualmente meno efficiente rispetto ai concorrenti Wasm, la possibilità di una compilazione AOT conferisce a WasmEdge eccellenti miglioramenti nelle performance.

WasmEdge, come dimostrato anche dai suoi plug-in, si dimostra quindi perfetto per operazioni computazionalmente onerose e con carichi molto elevati, come le operazioni di processamento delle immagini, oppure per operazioni molto veloci su una notevole mole di dati, come l'analisi di dati in tempo reale, utile per il machine learning e il deep learning.
In conclusione, pur presentando alcune limitazioni attuali, WasmEdge si configura come una scelta promettente e dinamica nel panorama dei runtime wasm, con un ampio potenziale di miglioramento.

Bibliografia

[1] Marco Abbadini et al. «POSTER: Leveraging eBPF to enhance sandboxing of WebAssembly runtimes». In: (2023). URL: https://cs.unibg.it/seclab-papers/2023/ASIACCS/poster/enhance-wasm-sandbox.pdf.

[2] Alfred V. Aho et al. *Compilers : Principles, Techniques, and Tools*. A cura di Michael Hirsch. Second edition. Greg Tobin, 2006.

[3] Stephen J. Bigelow. «WebAssembly». In: (2023). URL: https://www.techtarget.com/searchitoperations/definition/WebAssembly.

[4] CNCF. *WasmEdge*. URL: https://wasmedge.org/.

[5] «Comparison». In: (2023). URL: https://wasmedge.org/docs/start/wasmedge/comparison.

[6] Antonio Corradi e Luca Foschini. «Principles, Applications and Models for Distributed Systems M». In: (2019). URL: https://lia.disi.unibo.it/Courses/PMA4DS1718/materiale/CloudComputing.pdf.

[7] Alex Crichton. «Multithreading Rust and Wasm». In: (2018). URL: https://rustwasm.github.io/2018/10/24/multithreading-rust-and-wasm.html.

[8] «Docker overview». In: (2023). URL: https://docs.docker.com/get-started/overview/.

[9] Solomon Hykes. In: (2019). URL: https://twitter.com/solomonstre/status/1111113329647325185.

[10] Michael Irwin. «Introducing the Docker+Wasm, Technical Preview». In: (2022). URL: https://www.docker.com/blog/docker-wasm-technical-preview/.

[11] Tabassum Khanum. «Server-Side Rendering (SSR) Vs Client-Side Rendering (CSR)». In: (2021). URL: https://dev.to/codewithtee/server-side-rendering-ssr-vs-client-side-rendering-csr-3m24.

[12] «Layers». In: (2023). URL: https://docs.docker.com/build/guide/layers/.

[13] Charles E. Leiserson et al. «There's plenty of room at the Top: What will drive computer performance after Moore's law?» In: (2020). URL: https://www.science.org/doi/10.1126/science.aam9744.

[14] Jordon Mears. «Performance of WebAssembly: a thread on threading». In: (2019). URL: https://medium.com/google-earth/performance-of-web-assembly-a-thread-on-threading-54f62fd50cf7.

[15] Marco Prandini. *Sysadm nell'era del Cloud computing*. 2023. URL: https://virtuale.unibo.it/pluginfile.php/1563377/mod_resource/content/1/cloud-2022-2023.pdf.

[16] Marco Selvatici. «Webassembly Tutorial». In: (2021). URL: https://marcoselvatici.github.io/WASM_tutorial/ref/sandbox.gif.

[17] Second State. «WebAssembly Serverless Functions in AWS Lambda». In: (2021). URL: https://www.secondstate.io/articles/webassembly-serverless-functions-in-aws-lambda/.

[18] B Shyam Sundar. «WASM + WASI + WAGI + Web Assembly Modules in Rust». In: (2022). URL: https://medium.com/@shyamsundarb/wasm-wasi-wagi-web-assembly-modules-in-rust-af7335e80160.

[19] «The LLVM Compiler Infrastructure». In: (2023). URL: https://llvm.org/.

[20] W3C. *Wasm*. URL: https://webassembly.org.

[21] «WasmeEdge flow». In: (2023). URL: https://github.com/wasmedge/docs/blob/main/docs/contribute/internal.md.